AD-A154 624

Globally-Asynchronous Locally-Synchronous Systems

by

Daniel M. Chapiro

Department of Computer Science

Stanford University
Stanford, CA 94305

DTIC
SELECTE
JUN 3 1985
S   D
A

DTIC FILE COPY

85 02 07 062

GLOBALLY-ASYNCHRONOUS

LOCALLY-SYNCHRONOUS SYSTEMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Daniel M. Chapiro

October 1984

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.
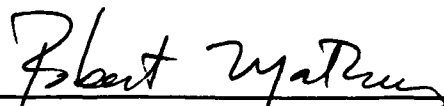
_____

Robert Mathews, Electrical Engineering.
Principal Advisor

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.
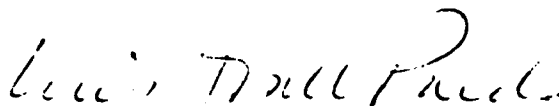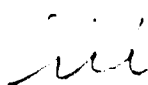
_____

Forest Baskett, Computer Science

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

_____

Luis Trabb-Pardo, Computer Science

Approved for the University Committee on Graduate Studies:

_____

Dean of Graduate Studies and Research

Globally-Asynchronous

Locally-Synchronous Systems
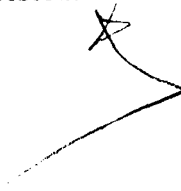
Abstract

This thesis provides a new framework for the design of very high performance digital machines. The new theoretical results which are presented have practical implications, and lead to a better understanding of possibilities and limitations in the design of computers, communication hardware and other digital machinery.

The discussion centers on different organizations for globally-asynchronous, locally-synchronous systems, and covers the following issues: organizations for complex digital systems, metastability as a limitation for high performance, structures for two classes of non-conventional architectures, optimization, performance, reliability, and design techniques.

We present new algorithms to compile the specifications of such machines onto efficient circuits, and to verify the correctness of the resulting machines. The models we developed for the analysis of the tradeoffs between different variables that affect the safety of operation of these systems, show that the proposed organizations result in extremely fast and reliable digital machines. The proposed organizational schemes can be used within a wide range of architectures, and integrated circuits designed according to this methodology have been developed and tested.

Acknowledgements

Contents

Figures and Tables

Introduction

Let us start by looking at the top level organization of a digital system. Complex
items have many components interacting with each other in different ways. In
nchronous systems, each of these components receives a common periodic signal
ock) that is used to control its operation and its interaction with other components.
wever, a typical computer will have components such as CPUs, disks, terminals, and
mmunication lines that clearly cannot be all tied to the same clock. Hence, complex
items are not designed as a single synchronous block, but as an ensemble of locally-
nchronous components interacting with each other without a global clock. When we
rtition a system into sub-components that do not share the same clock, we say that
ey are *asynchronous* respect to each other.

There is still another another reason that suggests further partitioning of the
mponc ts: When signals are transmitted, they take some time to arrive to their
stinatio. ting a limit on how far a datum can travel within a clock cycle. In
neral, the s. a synchronous component is, the faster it can be clocked.[1]

In conclusion, synchronous systems cannot grow in complexity without limit be-
use they have to interact with other components that cannot share the same clocking
ntrols, because of delays in the communication across a system, and because of clock
ews.

---

lso, within a synchronous block, clocking signals will arrive at the sub-components with possibly
fferent delays (this is called clock *skew*). The larger the delays, the harder it is to keep the skew
nall, and as the skew becomes larger (approaches the duration of the clock period), the synchronous
eration becomes more difficult. Hence, skew is another limitation to the size of synchronous blocks.

tter in that the extra edges may be needed anyway to provide time references for
t-up and hold times, de-skewing, and other purposes (*e.g.,* sending addresses before
.ta on a write operation), and it requires less logic in the communications mechanism.

For a more extensive treatment of communication problems, the reader is referred
[44].

!.7 Machines with Stretchable Clocks

The main motivation behind machines with stretchable clocks has been to avoid
e metastability problems we have discussed and to implement self-timed structures.
retchable clocks can stretch a clock phase for an unbounded period of time, but
inetheless continue with a normal gap and normal cycles immediately after the
retching, *i.e.,* clock cycles succeeding a stretched cycle are only displaced in time,
it not affected otherwise. Stretchable clocks are described in more detail in [39], and
so later in Chapter 4 and in the appendices.

Pechoucek proposes stopping the clock as long as a flip-flop remains metastable
7], or stopping the clock until an external event occurs. Stucki [43] shows in more
tail how asynchronous signals can be sampled with a synchronizer that detects
etastability, and how to use this information to stretch a clock cycle as long as may be
cessary for the metastable state to subside. In this way, Stucki avoids synchronization
ilures at the expense of infrequently allowing clock phases to stretch for unbounded
riods of time. Attempts to bound the stretching period do not work (see Chapters
and 4); for example, the suggestion in [43] about using a time-out re-introduces
nchronization problems that it had previously eliminated, moving the failure from
ie place in the circuit (the asynchronous sampling) to another one (the decision to
-sample the data when the time-out arrives).

Seitz Pipeline Modules [39] are particularly interesting in that they consist of
cally-synchronous machines communicating asynchronously in such a way that they
i not need synchronizers to handle that interaction, but still do not have synchroniza-
in problems. They use a 4-cycle handshaking protocol to communicate asynchronously
th other machines. The handshaking signals interact with a stretchable clock and
ay stop a local machine on a particular phase when what it needs from its neighbor
ocessors is not available, letting the clock continue when the resource becomes avail-
ile. Unfortunately, the only way of fully understanding why the SPM works is to

can grow linearly with α [39], while $1/\lambda_m$ and W decrease approximately linearly with α [42, 29]. If we maintain the size of the chip, and the number of synchronizers grows approximately as the number of devices along the periphery of the system, the *chip* reliability decreases approximately with the *square* of the scaling factor.

§2.6 Low-Level Asynchronous Communication Protocols

When two digital machines need to communicate, which forms of communication are possible depends upon the assumptions the designer can make about delays. For example, if we know nothing at all about the time it may take for another system to respond to a request, clearly we cannot use the approach used internally to synchronous systems, where a worst case delay is assumed, and after this time has elapsed it is implicitly known that whatever the system had to do is actually done. Instead, the systems must exchange signals to request tasks to be done, to indicate that tasks are completed, and also possibly to acknowledge the reception of some of the signals themselves. The particular way in which this exchange of signals may proceed is known as a communication protocol.

In a commonly used set of asynchronous protocols, the communicating parties exchange what can be looked upon as a polite handshake. For our purposes we are mainly concerned with 2-cycle (transition-sensitive) and 4-cycle (level-sensitive) handshakes. In both there is a *request* (*Req*) signal to request some action, and an *acknowledge* (*Ack*) for the request.

What distinguishes the protocols above is the way in which the signal transitions are used. The actual meaning of the *Req* and *Ack* signals is not given by the protocol. The request may in fact be a command, or maybe even a request for a new command; the acknowledge may indicate that the request has been satisfied, or that the request has been received; etc. The protocol only establishes the temporal order of *Req* and *Ack* : irrespective of the interpretation, a request always precedes an acknowledge.

For both 2-cycle and 4-cycle signaling, the *Req* and *Ack* signals make the same transitions in the same order to complete a full cycle: $Req^{\uparrow} \prec Ack^{\uparrow} \prec Req^{\downarrow} \prec Ack^{\downarrow}$, where the arrows denote rising or falling edges of signals. Their only difference resides in the fact that 2-cycle signaling uses the rising edges to accomplish one transaction (*e.g.*, a data transfer), and the falling edges to accomplish a second one, while 4-cycle signaling makes a single transaction with the complete cycle. The advantage of 2-cycle signaling is that it uses fewer signal transitions [39], and is therefore faster. 4-cycle signaling is

where λ_m is the probability of exit per unit time (decay rate). Experimental results confirm that this model is quite accurate [42,38, 12].

Assume we are sampling data that may make transitions at random times, and that the probability that a data transition will occur within a given clock cycle depends only on the duration of the clock cycle. Assume that as the clock period tends to zero, the probability that more than one transition occurs within the same phase tends to zero faster than the clock period. For any real application, we could assume that it becomes zero altogether below some clock period. These two assumptions are necessary and sufficient conditions [8] to know that the transitions are generated by a Poisson process $P_n(\lambda t) = \frac{e^{-\lambda t}(\lambda t)^n}{n!}$, where $\lambda = f_d$, which is the expected number of data transitions per unit time.

Call W the window of time around a clock edge where a data transition would trigger a metastable condition. For a Poisson process with rate f_d, the distribution of events over a bounded interval W is uniform [8], and the expected number of arrivals in such an interval is $f_d W$. Therefore, the probability of entering a metastable condition at the beginning of each clock cycle is:

$$P(met_{t=0}) = f_d W. \tag{2.2}$$

From equations (2.1) and (2.2), the probability that a given clock cycle will result in metastability that lasts at most a time t is:

$$P(met_t) = P(met_t|met_{t=0})\, P(met_{t=0})$$
$$= f_d W e^{-\lambda_m t}. \tag{2.3}$$

For conventional systems with fixed clocks, if the synchronizer is still metastable when the time t_r allotted for synchronization is exhausted, we say there has been a synchronization failure. Let f_c be the sampling clock frequency. The probability of failure in a single clock period is then $P(\text{fail in 1 cycle}) = P(met_{t>t_r})$. Since we assumed independent data transitions, the number of failures in n clock cycles will have a binomial distribution with an expected number of failures $n P(\text{fail in 1 cycle})$. Therefore, the MTBF for a conventional machine with a fixed clock period is:

$$\text{MTBF} = 1/E[\text{number of failures per unit time}] = \frac{e^{\lambda_m t_r}}{f_c f_d W}. \tag{2.4}$$

As integration and operating speeds increase, this failure problem becomes more relevant. Let α denote the scaling factor [39]. Within a reasonable range, f_c and f_d

reports of MTBFs of synchronizers built using newer technologies [12]. The following references were chosen based on their readability. Catt [10] analyzed the delays involved in the use of bistables for synchronization purposes. Chaney *et al.* [12] and Pechoucek [37] pointed out the risk involved in the metastable operation of flip-flops. Hurtado and Elliot [30] have shown that metastability of flip-flops is unavoidable under reasonable (but restricted) conditions. Barros and Johnson [9] have shown that given a perfect synchronizer (one that will never enter a metastable condition) one could make a perfect arbiter (one that will always produce an arbitration in a bounded amount of time) and vice versa. Veendrick has shown that noise does not decrease metastability risks [46]. Chaney has provided some recent extensive measurements of flip-flop response under marginal triggering [12]. The phenomenon involved is subtle enough that impossible devices to get around metastability [48] are sometimes proposed. Finally, for bistables, Marino's paper [32] is conclusive. It rigorously proves under very weak conditions that synchronous sequential circuits exposed to inputs that can change asynchronously with respect to the clock of the circuit cannot avoid metastable conditions.

The conventional solution to the metastability problem is to run a synchronous system slowly enough, so that the probability of failure is acceptably small. In some cases it is possible to pipeline several flip-flops, or use alternate synchronizers and multiplexers, instead of reducing the clock speed, to obtain an adequate synchronization time. Nonetheless, if the response time is critical, as is the case for most arbitration problems, pipelining will not help, and the system must be slowed down. In any event, such systems will occasionally fall prey to metastability during synchronization.

§2.5 Quantifying Synchronization Failure

The time it takes a flip-flop to exit from the metastable region is unbounded. Nevertheless, the probability that it remains metastable has been found by empirical methods to decrease exponentially with the duration of the phenomenon. There are theoretical models that explain this behavior as follows. Assume that a metastable flip-flop has no memory as to how long it has been in a metastable region, and that the probability of decaying to a stable state is time-invariant. The only distribution that is "memoryless" is an exponential distribution $1 - e^{-\lambda_m t}$ [8, 20]. Hence, the probability that a flip-flop remains metastable for a period of time t or longer, given that it was metastable at $t = 0$, is

$$P(met_t|met_{t=0}) = e^{-\lambda_m t}, \tag{2.1}$$

the flip-flop remains undecided, they are in a metastable state.

Why do we care about this situation? If we sample a signal, as in the figure below, we certainly want to use it some time afterwards (*e.g.,* in the next clock cycle). But, by the time we want to use it, there is no guarantee that the flip-flop will no longer be metastable.
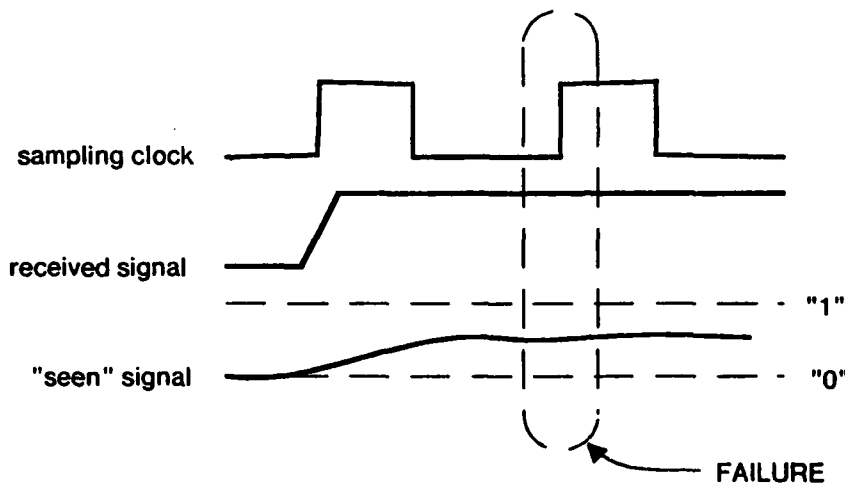
Figure 4. Synchronization Failure

When a flip-flop enters a metastable state, the probability of exit to some other state in a fixed time interval is very high, but not 1. The probability of sampling a signal and placing a flip-flop in a metastable state is very small, but is a practical concern [42]. In actual digital systems, asynchronous signals may be sampled at such a high rate, giving so little time for the synchronizer to evolve out of the metastability region, that a significant fraction of the synchronizations may not be completed, leaving the flip-flop in a metastable state. Thus, the system is still exposed to the same kind of inconsistent interpretations of the input data as a system without synchronizers.

There is abundant physical evidence produced at the Washington University of St. Louis and elsewhere [42, 37, 47, 38, 17, 12] that shows that metastability is a real problem, and there is also mathematical evidence [32] that suggests that metastability is an inescapable fundamental problem for any synchronous sequential system with asynchronous inputs.

A number of researchers have contributed to the related literature, which goes back to the 50s, when metastability was noted as a possible cause of transient malfunctions; the 60s saw a spurt of activity in this area, but currently it is dormant aside from

pointer but not read a character, or read a character but not advance the pointer, or give a green light to two intersecting streets, or open the air compressor intake but close the kerosene valve of a jet engine).

To avoid such problems, asynchronous signals are never used directly, but are first fed to a synchronizer, which is typically a D flip-flop or an equivalent regenerative circuit that is not stable at intermediate (digitally undefined) values. The external signal is sampled, held for a while in the flip-flop, and later used by the receiver.

The problem is that a flip-flop, like all bistable elements, has two potential energy minimae (the stable states) and a maximum separating the local minimae, as can be seen in the figure below.
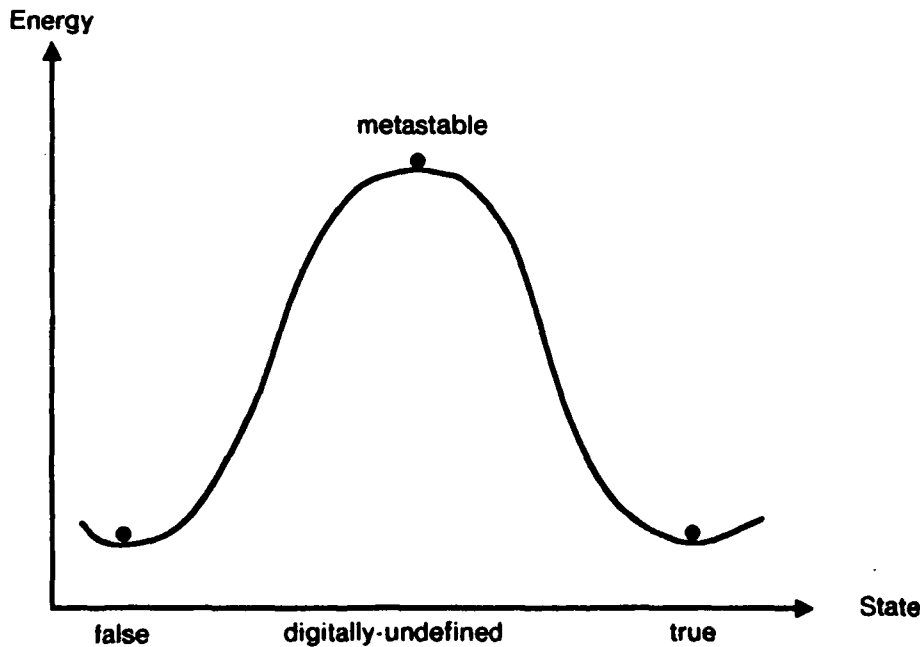
Figure 3. Energy of a Flip-Flop

Although any system will be stable only at the energy minimae, it can be metastable[4] at the energy inflection points. To make an analogy, imagine a pencil "perfectly" balanced on its tip, or a ball precisely located on top of a hill. If nothing disturbed it, it might stay there for a long while. As long as the ball remains on top of the hill, or

---

[4]When a system is metastable its state remains stationary, but any slight perturbation may push it away from this state.

completion signals have slowed the original designs as much as the self-timing has speeded them, with no significant net gain [6].

Nonetheless, the self-timed design paradigm has been influential in many different ways, as exemplified by Petri nets [3], data-flow machines [18, 22], variable-speed adders, self-timed modules [41], Muller's C-element-based modules, Seitz's self-timed modules [39], and Pechoucek's "fundamental solutions" [37], to name just a few. In all of these, the common feature is the ability to generate completion signals and to use these signals appropriately to control the sequencing of computations that take an amount of time that may not be known in advance.

Self-timed structures are quite diverse, but for our purposes, they can be classified along one important dimension, whose relevance will become apparent later: those schemes that use arbiters[3] or equivalent mechanisms, and those that don't. From the list in the previous paragraph, for example, Petri nets can represent machines that do have arbiters, and data-flow machines need arbitration mechanisms, but Seitz's pipeline self-timed modules (SPMs) don't use them.

§2.4 Synchronizers, Metastability and Synchronization Failure

Those self-timed machines that use arbiters, synchronizers, or other equivalent mechanisms face a substantially different problem from the ones that don't. Similar asynchronous interactions also occur when synchronous components sample asynchronous data or require an arbitration to get access to some resource. In this section, we will discuss what the problem is and the impact it has on machine organization.

When two synchronous systems are run from independent clocks and have to communicate with each other, they need to take special care in handling the signals received from each other. The reason is that since they do not share a common time reference, the receiver may sample what the sender sent precisely when the corresponding signal is changing. The receiver may get an intermediate value, which is digitally undefined. If it were to use that value without further ado, different components of the receiver might make inconsistent digital interpretations of the value, with the possible consequent failure of the receiver (e.g., it might advance a buffer

---

[3]An arbiter is a device capable of receiving requests for a unique resource, from multiple sources, and assigning the resource in the order in which the requests arrived. Most important, it can tell who arrived first.

example, it is possible to avoid some kinds of hazards[2] if we can guarantee that no more than one input can change asynchronously before the system has settled [45], or if several may change, if they satisfy minimum delay constraints [45, 21] between each signal change. In general, a machine connected to several other asynchronous machines must not require such guarantees because it cannot know in advance the times at which other machines may attempt to interact with it.

In general, because the restrictions we pointed above cannot be satisfied in practice, these classical methods are not applicable to the design of globally-asynchronous computers. Instead, designs have tended mostly to remain synchronous, using "synchronizers" at system boundaries, or have gone in the "self-timed" direction. We will discuss these two approaches in the following sections.

§2.3 Self-Timed Machines

The literature on self-timed machines abounds with interesting concepts.   It was recognized very early that it might be advantageous to have each component compute at its own speed and emit a completion signal on finishing whatever task it was assigned. Any component carrying out some task needs some time to complete it, and obviously we cannot ask for the results before this time has elapsed. In synchronous systems, a worst-case assumption is made about how long it might take to complete the task, and this time is measured by a centralized clock.

In contrast, in a self-timed structure, the component that performs the task also indicates when it has finished, thereby allowing other components to use the results right away, instead of always waiting for the worst-case time. The advantage is obvious: self-timed machines can run at a speed related to the average case, instead of the worst case. The disadvantage is that each component must have extra circuitry (a) to compute its own completion signals, and (b) to check for completion of requests it may make to other components. These completion signals, which are exchanged to control the sequencing of operations, are called *handshaking signals*.

Most machines that have been designed with this approach have failed to realize the expected improvements, because the circuits required to compute and handle the

---

[2]When a signal that should maintain a constant value glitches briefly, we say there is a "static hazard", and when a signal that should have made a single transition bounces before settling to its new value, we say there is a "dynamic hazard" [45].
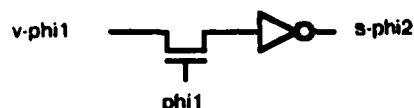
Figure 1. Making a valid signal stable

If all the inputs to a combinational logic block are s-φ_1 , the outputs are also s-φ_1 . If some (or all) the inputs are v-φ_1 and the rest of the inputs are s-φ_1 , the outputs are v-φ_1 . To generate a q-φ_1 signal, AND a s-φ_1 signal and φ_1 .

Figure 2. Propagation of Signal Types

§2.2 Classical Approaches to Asynchronous Design

It would be very nice if we could stay within the relatively simple world of synchronous systems, but as we saw in the introduction, we cannot run disks, communication lines, terminals, controllers, etc., all from the same clock. In consequence, we must analyze how components talk to each other when they are mutually asynchronous (*i.e.,* when they have independent clocks).

Classical asynchronous design focuses on extending the methods that are used for synchronous machines to the asynchronous domain. The results in this field are abundant and very interesting, and a thorough coverage can be found in Unger's [45]. Nonetheless, they generally focus on design problems at a very low level of integration, and put severe restrictions on the kind of asynchronous signals they handle. For

2.1.1 Basic Strict Types

Assume that there is a two-phase (φ_1 and φ_2), non-overlapping clock, as is common in MOS designs [34]. For brevity, the definitions of the signal types refer just to φ_1, but of course the same holds for φ_2. All signal types are referred to the rising and falling edges of the clock phases ($\varphi_1^\uparrow$, $\varphi_1^\downarrow$, $\varphi_2^\uparrow$ and $\varphi_2^\downarrow$).[1] The types of interest are:

- A signal X is validφ_1 (Xvφ_1) if it settles in a bounded period after $\varphi_1^\uparrow$ and remains unchanged at least until $\varphi_2^\downarrow$. If we increased the length of the phase φ_1, X's settling would stay anchored to $\varphi_1^\uparrow$.

- A signal Y is stableφ_1 (Ysφ_1) if it settles in a bounded period after $\varphi_2^\downarrow$ and remains unchanged at least until the next $\varphi_2^\uparrow$. If we increased the length of the φ_2-φ_1 gap, Y's settling would remain anchored to $\varphi_2^\downarrow$.

- A signal Z is qualifiedφ_1 (Zqφ_1) if it can only be asserted no later than a bounded period around $\varphi_1^\uparrow$ and is cleared no later than a bounded period around $\varphi_1^\downarrow$ (*e.g.*, it can be generated ANDing a stableφ_1 signal with φ_1, in which case it will follow the rising and falling edges of the clock with a delay bounded by the time it takes to AND the clock and the stable signal).

- A signal W is valid-qualifiedφ_1 (Wvqφ_1) if it is generated by ANDing a validφ_1 signal with φ_1. Wvqφ_1 is like Wqφ_1, with the exception that it may glitch during a bounded period around $\varphi_1^\uparrow$.

Note that these relations must hold independent of the length of clock phases.

2.1.2 Some Properties and Composition Rules

Let a φ_1-$\mathcal{ME}$ be a memory element that receives its input during φ_1. Its input must be at least of type v-φ_1 (it can be s-φ_1), and the sampling control signal must be at least q-φ_1 (it can be φ_1). A φ_1-$\mathcal{ME}$ will have a s-φ_2 output. A dynamic nMOS memory element appears in the figure below, where a pass transistor samples a v-φ_1 signal and loads a capacitive storage node during φ_1. The output of the inverter will be s-φ_2, and will change again only with the rising edge of the following φ_1.

---

[1] For brevity, both the notation and the results of [36] are presented here in a slightly different and simplified way.

Background and Previous Work

This chapter provides the necessary background for understanding the issues discussed in this dissertation, making it more self-contained. It is assumed that the reader already has some familiarity with digital system design.

There have been numerous approaches to the design of digital machines, each focusing on some particular problem. We will start by discussing a synchronous clocking discipline that we can use for our locally-synchronous components. Then we will see why the classical approach to asynchronous design is not useful for our globally-asynchronous structures. Subsequently we will discuss issues relevant to interaction between machines that do not share a common clock; these issues are self-timing, synchronization failure, and some asynchronous communication protocols. The chapter concludes with a review of some proposals for the use of stretchable clocks for the implementation of self-timed machines, which avoid synchronization failures.

§2.1 A Synchronous Design Methodology

The problems of synchronous design are, in general, well understood. For our locally-synchronous modules, it is convenient to borrow the structure and notation of some synchronous discipline. We will use the strict two-phase clocking discipline of Noice [36] (although we could as easily use any other reasonable scheme) because although we will focus more on the global non-synchronous interaction, we will still need a consistent notation to describe the interactions between the non-synchronous machinery and the synchronous components, and among the synchronous components themselves.

to compile them into circuits.

We chose to put all the detours, no matter how interesting or subtle, in appendices, so that the core of the ideas is not "hidden by the trees". Generally, the appendices develop some central ideas in greater depth, but are not necessary to get the overall picture or to understand the main body of the thesis.

- Lack of a unified general approach, which is evident in bugs in some published circuits [48] and in some interesting (but hard to modify) locally-synchronous, globally-asynchronous machines [39].

- Area inefficiency in completely self-timed approaches, with the consequent speed penalties.

- Overly restrictive design disciplines that require some kind of functional module for any operation, no matter how simple it may be.

Therefore, an attempt is made here to solve at least part of those problems by developing a practical, unifying theory that is flexible and general, allowing a more systematic design of reliable high-performance machines.

§1.1 Scope and Outline

This thesis presents a unifying theory that leads to a better understanding of possibilities and limitations in the design of digital machines. The core of the thesis is a new set of general organizations for reliable, high-performance, globally-asynchronous, locally-synchronous machines, and methods for analysis and synthesis of these machines. A probabilistic model and calculations for speed and reliability compare them with conventional machines. New algorithms allow us to transform high-level specifications of these machines into area-efficient and time-efficient circuits that can never have metastability problems.

Chapter 2 provides background on digital design, on synchronization and metastability, on self-timed machines, on asynchronous communication protocols, and on machines with stretchable clocks.

Chapter 3 explores digital communications, discusses the reasons for the structures we will study, lays down the basic assumptions, introduces the "value-safety" and "time-safety" concepts, and develops a new characterization for computing machines that opens two basic paths that are explored in Chapters 4 and 5.

Chapter 4 explores the class of "unsynchronous" machines, explains how they are constructed and why they attain value-safety. Then, it analyzes their performance and reliability, and compares it with that of conventional machines.

Chapter 5 describes the class of "escapement" machines, discusses their performance, analyzes optimizations, and presents algorithms to verify their value-safety and

Since everything points in the direction of breaking synchronous systems into smaller systems interacting asynchronously, a natural question arises: Why not break them all the way down to their smallest components? The answer can be found by examining communication costs. Sub-components of a synchronous systems can communicate in simple ways by agreeing on the time windows in which one will send some datum to the other, but if these sub-components do not share a common clock, they have to resort to more complex (in time and area) mechanisms to transfer the same datum. Ultimately, if we partitioned a system all the way down to the simplest possible sub-components (all interacting asynchronously with each other), the communication mechanisms between elements would dominate the space and time used for the actual computation.

As a consequence of these two opposing factors, there is an optimum middle ground for the structure of a system where we do not have a monolithic synchronous system, but also we do not partition the system all the way down into its simplest components. The synthesis and analysis of such *globally-asynchronous, locally-synchronous* systems is the focus of this thesis. The particular aggregation level at which we will stop the partitioning is technology and application dependent. Nonetheless, to give a feeling for the kind of structures we will study, we can give some loose bounds: assume that the synchronous components may range in size from about a thousand transistors (possible a part of a chip) up to what might fit in a rack.

With the advance of device technologies, faster and bigger computers are being built, and the costs are shifting from the active elements to the communication lines. Consequently, researchers have attempted to partition systems in ways that allow each component to run at speeds limited only by their internal structure, and not by the overall system size [45, 39]. Measured by their acceptance in industry, most of these attempts have failed, and only part of this failure can be attributed to industry's inertia or to the "not invented here" syndrome.

We will not attempt a critical survey here, but to understand some of the reasons behind prevalent design practices, it is worth citing a few problems in previous approaches:

- Overly restrictive asynchronous design methods, in which the restrictions that are put on how or how many signals may change makes the design of complex systems virtually impossible and extremely cumbersome [45, 21].

stare at the circuit and draw timing diagrams. A careful delay analysis is required to implement this machine.

These machines with stretchable clocks implement Pechoucek's two "fundamental solutions" [37], and can be looked upon as two points in a design space that we will explore in depth in Chapters 4 and 5.

Machine Organization

In the introduction we discussed the reasons for our focus in machines that have a globally-asynchronous, locally synchronous, (GA-LS) structure. Here we discuss two particular kinds of reliability, value-safety and time-safety. Then we propose a taxonomy of machines that is based mainly on how the completion of activities is handled. The taxonomy provides a framework for the analysis of GA-LS architectures, which is useful for understanding and designing high-performance GA-LS machines.

We saw in the introduction that there are strong reasons for partitioning high performance machines into synchronous clusters interacting asynchronously with each other. The appropriate size for each synchronous cluster is technology-dependent and problem-dependent, and should be determined by the architect of the machine for each specific case. In the introduction we proposed reasonable upper and lower limits, but our analysis is independent of granularity and details of system partitioning. We simply provide ways to make these machines communicate quickly and reliably with each other.

In this Chapter we define the kind of synchronization problems that we must face when designing hardware, to guarantee that the next higher abstraction level [44] will operate with Boolean values defined over a discretized time. Note that traditional software synchronization problems [27] *assume* a completely digital universe and a discretized time, which is inadequate for hardware, and . their solutions do not extend to hardware synchronization.

We analyze how digital machines handle completion signals, and develop a taxonomy based on the assumptions the designer makes about these completion signals. We

present the value-safety and time-safety concepts, and two new theorems that focus
our research on GA-LS machines on 2 classes of GA machines. These classes explored
in Chapters 4 and 5.

§3.1 Completion Handling

What follows describes the hardware design problem that we consider and the
basic assumptions we make. We will use this conceptual framework together with the
value-safety and time-safety concepts to analyze the ways in which the asynchronous
interactions can take place.

3.1.1 Computation Model

We want a general model applicable to any kind of machine, independent of its
particular architecture. Clearly we should be satisfied to be able to compute anything
that can be computed with a Turing Machine [31]. Partial recursive functions can
compute anything that is Turing computable. We choose this formalism because it is
rich enough to compute anything our hardware can compute (without any necessary
similarity with the actual hardware structure), and allows us to reach general conclu-
sions without being distracted by the details of how computation, memory, and timing
are related in the hardware implementation.

Assume we model our computations with such functions, mapping elements X
from an alphabet $\Sigma = \{0, 1, \upsilon\}$ onto itself, where 0 and 1 are boolean values and υ
is a value that is digitally undefined. Since our concern is for real machines, not only
logical constructs, we must specify how variables are represented physically. Assume
that variables are represented by some continuous physical parameter V. Furthermore,
assume that this parameter is a function of continuous time t. It is necessary to have
a physically realizable mapping from the variables representation onto our computing
alphabet, and there are many obvious possibilities for this (e.g., $V > V_{high\ min} \mapsto 1$,
$V < V_{low\ max} \mapsto 0$ and $V_{low\ max} \leq V \leq V_{high\ min} \mapsto \upsilon$).

For the sake of design convenience, assume that our computing machine is broken
into a number of functions $f_i(X)$, and that these functions take recursively or iteratively
results from previous invocations. Assume that any such invocation takes a time $\delta > 0$.
δ depends primarily on the function f_i that we are computing, but may also depend

on the values of the arguments, or even on the state of the whole machine. This time, which we call the *completion time*, may or may not be known to some given precision, and may or may not be bounded. If we attempt to use the result of a computation before it is completed, we assume that the result may not be defined, and we assign it the conventional value of v [31]. Note that although by using this model we do not need to talk about memory and timing in the sense used by clocking disciplines [36], we have introduced timing in a restricted sense when we partitioned the machine into many functions, each of which produces results used at a latter time by other functions.

3.1.2 Uses of Completion Knowledge

Assume that we are interested only in final results defined over the sub-alphabet $D = \{0, 1\}$. The basic problem is that even if we supply digitally defined inputs, we need to know when each computation has been completed so as to proceed with the next one. The way in which we are going to obtain completion information opens a number of design alternatives that depend on what we know about the delays and what use we want to make of this knowledge:

(a) If we know the values of all delays, we may choose to define intervals of time over which we guarantee that the inputs to succeeding functions become digitally defined. Using this information together with the structure of the machine we can calculate the times at which the computations will be completed, and control the flow of intermediate results through the machine. Note that this knowledge of the delays allows for the tightest timing, but a timing scheme that took advantage of all this information could be very complex.

(b) If the delays are not precisely known, but we have bounds on the time it may take to compute each function, we may choose to give every computation the time needed in the worst case by the slowest function. Clearly, a synchronous machine can provide this kind of control, and its clock implicitly generates completion signals.

(c) We may not have any bounds on the delays, but still need to provide completion signals that indicate when the inputs to functions are available [6]. This last case divides into several sub-alternatives:

(c.1) Each function block may provide an explicit output signal indicating when its own computation has been completed. Most typically, this kind of mechanism appears

in self-timed machines.

(c.2) We may have no bounds on the delays, nor a completion signal. The conventional approach to this problem is to assume a special function (a synchronizer), that can take a digitally undefined value and usually produce a digitally defined value in a bounded amount of time. For digitally defined values it is an identity function; otherwise it assigns randomly a 0 or a 1, or with a much smaller probability a v. The structure can be designed so that the results are not affected by the occasional random digital (0 or 1) assignments (by using appropriate redundant information in the communications protocols), and we will get mostly correct digital results. This approach puts this structure within the class of those synchronous machines that handle asynchronous inputs with conventional synchronizers.

(c.3) Under the same conditions as in c.2, we may use another special function that is also a synchronizer, but is different from the previous one in that instead of probabilistically assuming the completion of the synchronization, it generates an explicit completion signal [43, 1]. This approach puts this structure within the class of self-timed machines.

None of these organizations dominates the others in every aspect and application. This being the case, the big question is: How and when can we combine these organizations, what are the limitations of such hybrids, what structures are appropriate to implement them, how can we guarantee their correct operation, and what advantages can we get from them? Before answering this question, we will focus it more by describing fundamental problems in asynchronous communication. We constrain the possible answers by introducing two new theorems and a taxonomy that will guide our answers.

§3.2 Value-Safety and Time-Safety

It has been shown under very general conditions that any sequential synchronous system subject to asynchronous inputs is liable to enter metastable states [32], which may lead to unavoidable system failure. It is important to note that this is not only an interesting theoretical problem, but also a very practical concern, and failures in design of interfaces, as well as experimental results, attest to this fact [17, 47, 12, 42]. More generally, we characterize the problem formally by stating the following theorem (which is proved in an appendix):

DRV Theorem: *It is impossible to make a Boolean decision about a continuous*

value with a finite precision instrument in a bounded amount of time.

This theorem suggests classifying a computing machine along 2 dimensions: (a) whether it always manages to make Boolean decisions, and (b) whether it takes bounded amounts of time to produce its results. We call the machines that can guarantee their results to be digitally defined *value-safe*, and the machines that can guarantee the boundedness of their completion time *time-safe*.

§3.3 A Taxonomy Based On Completion

The notion of completion and how we handle it is central to our approach. There are two key features that we may or may not know (or choose to use) about a completion signal. (a) We may not know *when it happens* in relation to our local notion of time. For example, for signals internal to a synchronous machine, we know when they may change respect to the local clock, but we cannot know at what time in relation to our clock an interrupt may come. (b) We may not know *what happens* at completion. For example, for an asynchronous request following a 2-cycle communication protocol we know in advance that when it arrives, its corresponding line will become asserted and will not be de-asserted until it is acknowledged. When sampling a signal that comes from an unclocked A/D converter, we don't know in advance what value it will have at the completion of a sampling interval nor when it will change again. These possibilities give rise to four kinds of machines, shown in the figure below.

Figure 5. A Taxonomy Based on Completion

Class (1,0) contains all globally synchronous machines. Class (0,0) contains among

others the machines considered by asynchronous design in [45, 21], arbiters [9, 39] and synchronizers [9, 42, 32]. Class (1,1) is clearly uninteresting, since the completion signal carries no information (we know in advance both when will it arrive and what its value will be). Class (0,1) contains some kinds of self-timed machines, but not those containing arbiters [39], which are partially in class (0,0). We are going to focus on value-safe machines of class (0,0) and class (0,1), which we call *unsynchronous* and *escapement* machines respectively.

Note that this taxonomy does *not* establish classes along traditional synchronous/asynchronous or hetero-timed/self-timed lines. No existing terms exactly match the taxonomy, which addresses more fundamental aspects of communication than traditional taxonomies. Moreover, words like "asynchronous" and "self-timed" have already acquired many meanings.[1]

At this point we introduce a corollary of the Uncertainty Theorem that further constrains our solutions (a formal proof of this new theorem appears in the appendices): *No unsynchronous system can be both value and time safe. That is, if it knows with certainty the values with which it operates, then it cannot know with certainty the time.*

Clearly an unsynchronous machine that has both properties is precluded by the theorem above, but machines that satisfy one *or* the other property at least are not precluded. We are interested in those machines that are value-safe, or that are at least extremely value and time reliable (*i.e.*, mostly safe, but failing with a known low probability).

The importance of these distinctions and of studying each class separately will become apparent later, when we propose a practical design methodology for high-performance machines in the unsynchronous and escapement classes. Currently, there are few proposals and fewer designs in each of these two classes [37, 39, 43], and they are clearly outside the technological mainstream, in part due to performance and design problems that we attempt to solve.

§3.4 Summary

We have analyzed some problems related to the communication of digital infor-

---

[1]For example, asynchronous design, referring to designs without a clock, asynchronous processes in synchronous machines [27], asynchronous signals entering a synchronous system, asynchronous meant as self-timed in [6], self-timed in the restricted sense of [39] and self-timed in general, etc.

mation and presented a taxonomy for digital machines. This taxonomy makes reference to the way in which a system exchanges information. In particular, if such exchanges are asynchronous, we can have different amounts of information about the asynchronous signals. *Unsynchronous* machines are value-safe machines that don't know in advance anything about the values of the signals they receive. *Escapement* machines are value-safe machines that know in advance the values of the asynchronous signals they receive, and also know that these signals will not change until acknowledged. The Uncertainty Theorem precludes unsynchronous machines that are both *value-safe* (do not get confused with the values with which they operate) and *time-safe* (know time with a bounded error), and still have arbitrary asynchronous inputs.

The next two chapters deal with unsynchronous and escapement machines. We will use the framework presented in this Chapter to develop and analyze different organizations for value-safe GA-LS machines.

Chapter 4
Unsynchronous Systems

In the previous chapter we defined unsynchronous systems as value-safe, globally-asynchronous, locally-synchronous (GA-LS) machines that interact at the global level by exchanging asynchronous signals whose values the recipient does not know in advance, and whose transitions may occur at arbitrary times in relation to the recipient's clock. We study unsynchronous machines before escapement machines because they are structurally simpler and because the properties of the asynchronous signals they exchange are simpler.

This chapter starts by discussing how to design unsynchronous, GA-LS, value-safe machines. To make the discussion clear, we introduce a pair of building blocks: a stretchable clock and a synchronizer with a metastability detector. We describe them functionally here; detailed circuits appear in the appendices. These blocks are used to build unsynchronous machines, which are proved to be value-safe. Finally, we develop models to calculate the performance of unsynchronous machines.

§4.1 The Unsynchronous Mechanism

Unsynchronous machines must be able to deal with external signals that may make transitions at arbitrary times in relation to the local clock. Such asynchronous signals do not need to satisfy any particular protocol, and we will call them *unsynchronized signals*. In contrast, asynchronous signals also include those about which we may know in advance in which direction they will make a transition. We denote an unsynchronized signal X as $X_{uns y}$. An unsynchronized signal may change value at arbitrary times, in

arbitrary directions. Such a signal may have a digitally undefined value at any time. Therefore, if we sample it without any modification (*e.g.,* with a multistable latch that simply samples a value and holds it, but does not modify it), and then attempt to use it, sometimes we will use a value that is digitally undefined, making the system *not* value-safe.

Note that some higher level protocol may require that X_{unsy} not be withdrawn until some other event happens. Such restrictions are not necessary to guarantee value safety for an unsynchronous machine.

To capture an unsynchronized signal as a digitally defined value requires a synchronizer; *i.e.,* a regenerative bistable circuit that will be stable only for digitally defined values. Unfortunately, as we saw in Chapter 2, such a device will yield digitally undefined values with non-zero probability when given a bounded amount of time for synchronization, resulting in different digital components of the receiving machine making inconsistent interpretations of the value, possibly leading to failure of the system. Therefore, we must explore alternatives. Since what we want are value-safe GA-LS machines, we are forced to allow an unbounded amount of time for the synchronization and to use a synchronizer that will indicate explicitly when it has completed the synchronization, as suggested by Pechoucek [37] and by Stucki and Cox [43]. Therefore, we need a circuit that will be able to detect metastability and also a means for waiting for the completion of the synchronization without reintroducing a new synchronization problem. In the next section we provide a functional description of such circuits.

§4.2 Building Blocks for Value-Safe Circuits

Before we talk about the notation and methodology for designing unsynchronous systems, we will describe functionally some simple but useful basic blocks: a variable-speed, 2-phase, stoppable clock and a synchronizer with a metastability detector. Knowing the function of the blocks makes it much simpler to explain the unsynchronous machines, even if afterwards one does not use precisely these basic blocks.   More detailed information, as well as an nMOS implementation, can be found in the appendices. Note that once the blocks are designed and their timing verified, we can provide a higher-level functional description of the corresponding circuits, which need not refer to any internal or external delays.

4.2.1 A Variable-Speed, 2-phase, Stoppable Clock

Designs for clocks that will stretch a phase for unbounded periods of time have been proposed by Seitz [39]. The stretching signal is asserted synchronously with a phase, and de-asserted an arbitrary period of time later. Next we provide a functional abstraction for one particular design.

The clock generates two non-overlapping phases (φ_1 and φ_2), and has in addition two inputs ($stretch_{\varphi_1}$ and $stretch_{\varphi_2}$) for stretching the φ_1 and φ_2 phases respectively. In the absence of stretching, phases have a length determined by an external analog control.
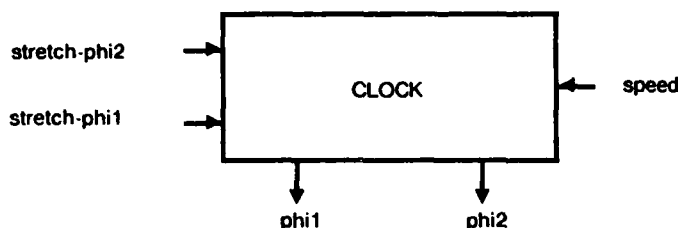
Figure 6. A Variable-Speed, Stoppable Clock

The stretch lines should be asserted synchronously with the phases of the clock and cleared asynchronously by some other process. Stretch signals must rise within a bounded period around the rising edge of the phase they will stretch (*e.g.*, a $stretch_{\varphi_1}$ may rise as a q-φ_1 signal or as a vq-φ_1 signal). The falling edge of a stretch signal, which indicates that there is no longer a need for stretching the phase, has no restrictions as to when it may come.

As long as the corresponding stretch signal is asserted, the corresponding phase will not terminate. A stretch signal does not change the length of ensuing gaps and phases; it just displaces them to the right on the time axis. A stretch signal that falls before the phase would have normally ended produces no stretching.

4.2.2 A Synchronizer with a Metastability Detector

The synchronizer shown in the next figure generates an explicit completion signal when it has finished the synchronization. It operates under the same principles as Seitz

rbiter [39, 40]. It receives two inputs: asynchronous data upon which there are no iming restrictions whatsoever, and a q-φ_1 sampling signal. It produces two outputs: a synchronized data signal which is s-φ_1 and a stretch-φ_2 signal. The data becomes available on the clock cycle following the one in which it is sampled.

Sample (q-phi1)

Asynchronous Input → **Synchronizer** → Synchronized Output (s-phi1)
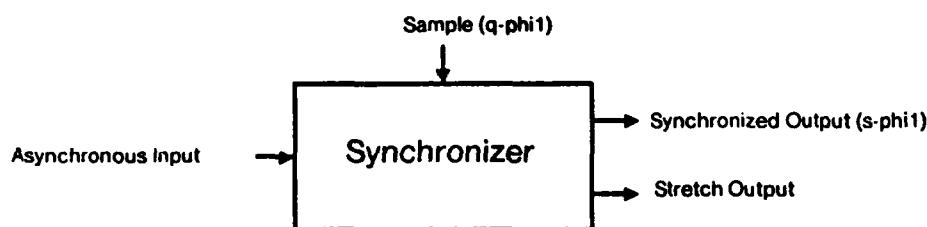
→ Stretch Output

Figure 7. A Synchronizer with a Metastability Detector

The stretch output may glitch harmlessly during φ_1 (we will never use it during φ_1), while a value is being sampled. The stretch signal may stay high during φ_2 as long as the synchronization is not complete yet. The stretch signal is guaranteed not to rise during φ_2: it can be high from before $\varphi_2^{\uparrow}$, and fall before or after $\varphi_2^{\uparrow}$, but once it falls, it stays low throughout φ_2 and up to the next $\varphi_1^{\uparrow}$. The stretch-φ_2 output from the synchronizer must be able to stretch the φ_2 phase for an unbounded period of time to guarantee that the synchronized data be digitally-defined [37, 6, 39] at the end of the synchronization period.

§4.3 Structures for Unsynchronous Systems

The general structure of unsynchronous systems is a global ensemble of locally synchronous machines (LMs), interacting asynchronously with each other through synchronizers that provide completion information. Since the machines are locally synchronous, a possible way of giving an unbounded synchronization time is to have the machine latch a value in one clock cycle, and use it in the following cycle, with the following cycle "arriving" only when the received signal becomes stable, as shown in the next figure. Connecting the output of the metastability detector to the "stretch" control of the clock assures that when the purely synchronous part of the machine attempts to use a value, by construction, it must be digitally defined.
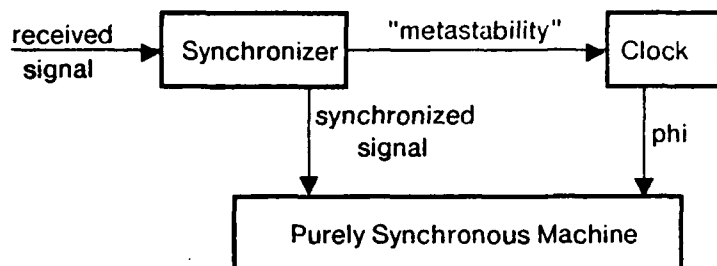
Figure 8. Block Diagram of an Unsynchronous Structure
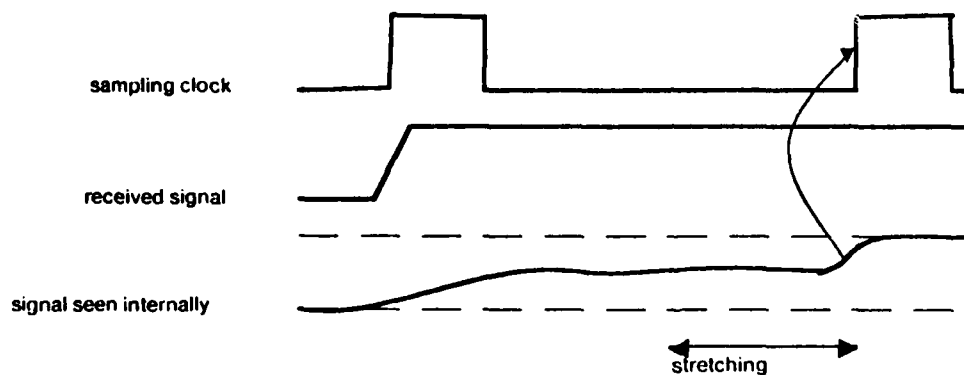
Figure 9. Waiting by Stretching a Clock Phase

It is important to note that there are theorems that preclude the design of unsynchronous machines that are value-safe *and* time-safe. It is tempting to "improve" the structures we propose in "harmless" ways so as to avoid the possibility of stretching the clock for unbounded periods. Nonetheless, such modifications invariably involve subtle bugs, since such machines, unfortunately, are as impossible as perpetual motion machines.

Although there are no theorems that preclude the value-safety of unsynchronous machines, nothing says that dropping the time-safety requirement actually produces value-safety. Therefore, it is important to show that we have not somehow hidden a synchronization problem in the metastability detection or in the stretchable clock. Next is a constructive proof that unsynchronous machines are value-safe.

a) A pulse generator (PG): The PG emits a single pulse of fixed duration when put falls. The output is produced by delaying the input and inhibiting the output the input is high, as shown in the next figure:
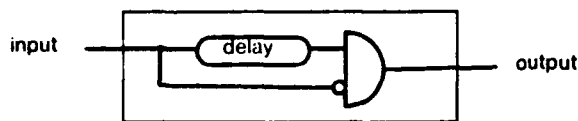
Figure 10. A single pulse generator

(b) A clock module (CM): A CM produces a single output which is the OR of a ch input signal and the output of a pulse generator (PG), as shown in the next e.
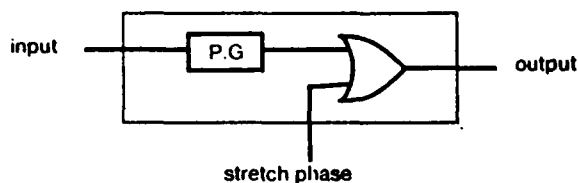
Figure 11. A clock module

(c) A stretchable clock: Put 2 CMs in a ring, feeding the output of each one to the t of the other, with an intervening delay in between each CM. If the stretch lines in low, the output of each module will trigger the other one, and their outputs be a sequence of non-overlapping alternating pulses (φ_1 and φ_2). To start the lation, pulse either of the stretch lines.
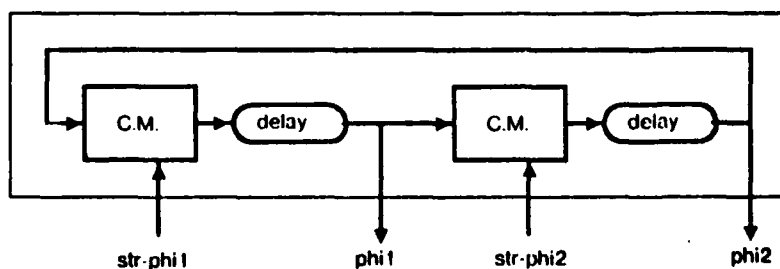
Figure 12. A stretchable clock

d that at the required clock rate, its MTBF is just 0.5 sec. If we use an unsyn-
ous machine instead, we know that the MTBF will climb to ∞. The question
such a huge reliability increase, what performance loss is acceptable? Using the
ion above and a typical value of $\lambda_m = 500MHZ$ for 4μ nMOS, $L = 10^{-6}\%$! Thus,
chronous machines can work with absolute reliability and without a speed loss
ditions so severe as to render synchronous designs virtually useless.

The Unsynchronous Limiting Speed

since unsynchronous machines cannot suffer synchronization failures, it is reason-
to ask how they respond if we crank up the clock. The propagation delays within
urely synchronous part of a given LM set a limit to the frequency of the clock.
ver, synchronization effects impose another limit. Since increasing the nominal
ency increases the number of stretched cycles, there is a limit to the average speed
iich the system will run. This limit is shown qualitatively in the figure below, and
ill now derive it quantitatively.
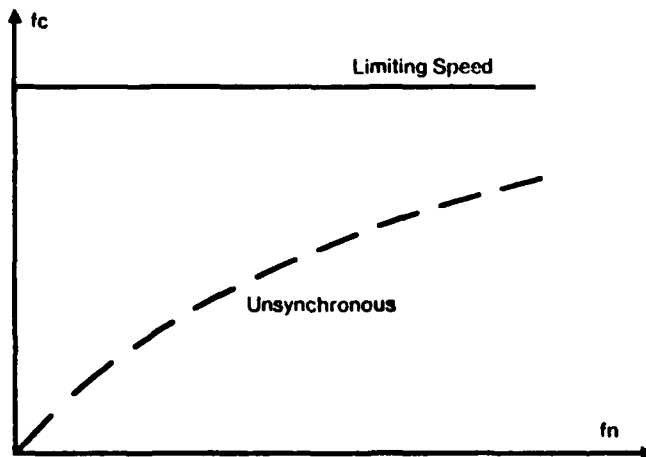
Figure 19. Limit Speed for Unsynchronous Systems

et d be the duty cycle of the clock, f_n the base rate of the stretchable clock that
to the system, and f_c the actual frequency at which the system is running, taking
account the stretching. Then:
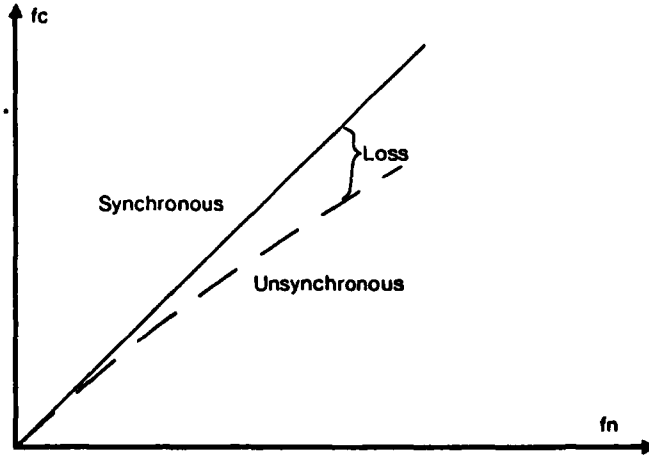
Figure 18. Speed Loss for Unsynchronous Systems

Next we derive the function $L(f_n)$ for the unsynchronous machine. From equation (4.4):

$$L_{unsy} = 1 - \frac{1}{1 + \frac{f_d f_n W}{\lambda_m} e^{-\lambda_m t_r}} = \frac{1}{1 + \frac{\lambda_m e^{\lambda_m t_r}}{f_d f_n W}} . \qquad (4.6)$$

Note that one part of this equation is identical to the MTBF_{syn} of a synchronous machine being clocked at the same frequency f_n, so:

$$L_{unsy} = \frac{1}{1 + \lambda_m MTBF_{syn}} . \qquad (4.7)$$

Since $1/\lambda_m$ is the expected time for a flip-flop to exit from a metastable state, $MTBF_{syn} \gg 1/\lambda_m$, so $_{syn}\lambda_m \gg 1$. Therefore,

$$L_{unsy} MTBF_{syn} \approx \frac{1}{\lambda_m} . \qquad (4.8)$$

This equation is very interesting because it links the reliability of a synchronous machine with the throughput of an unsynchronous one that performs the same task, both clocking at the same speed, and being perturbed asynchronously at the same rate.

To understand the implications of this result, assume that we were to perform the following experiment. We build a synchronous machine in 4μ nMOS technology, but
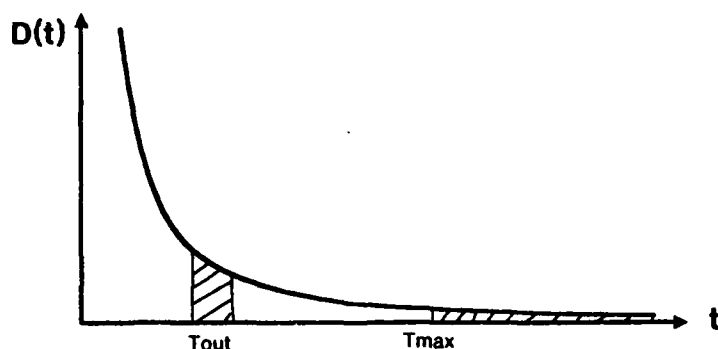
Figure 17. Bounded vs Unbounded MTBF

Therefore, the probability of having the UNSY machine fail to respond quickly ough may be smaller than the probability of the UNSYB machine having a synronization failure. Since bounded stretching requires more complicated circuits, we e better off with a simple unsynchronous machine than with the hybrid UNSYB chine, whose reliability is not clearly superior.

6.6 Speed Loss

The throughput equation (4.4) provides a measure of the speed of an unsynronous machine. Define the loss of throughput as $L = 1 - T$, which is zero for nchronous systems. If we plot f_c, the average frequency of the clock, as a function of , the nominal frequency of the clock, we get a straight line at $45°$ for a synchronous stem. For an unsynchronous one, the faster we drive it, the higher the percentage cycles that will be stretched, and the higher the throughput loss. The qualitative havior is shown in the figure below.

With a timer, the UNSYB is no longer value-safe because the timer's output is unsynchronized with respect to the disappearance of metastability in the synchronizer. This race is dangerous: if the synchronizer remains metastable throughout the full time-out period, then begins to resolve towards True precisely when the timeout mechanism is trying to steer the result to a logically defined value (False), the system may fail. Note that if the synchronizer resolves before this critical window, there is no problem. If it attempts to remain metastable longer, the value is cleared cleanly by the timeout mechanism. Only if it tries to resolve towards True in a very narrow window W exactly after the full timeout period will there be a synchronization failure.

4.6.5.2 Evaluation of Unsynchronous Machines with Bounded Stretching

We compare UNSYBs with UNSYs and with conventional synchronous machines (SYN). The comparison with SYNs is straightforward. Suppose that a SYN running at a clock speed f_c samples a datum in each clock cycle and pipelines its synchronizations through k stages to improve reliability. Suppose that the UNSYB also runs at f_c, but has no pipelining of synchronizations, and that $T_{out} = q f_c$. By making $q - k$ big enough, say 20 clock cycles, the UNSYB will be much more reliable than the SYN, and simpler too, since it does not use pipelining. The disadvantage is that the UNSYB will have a worst-case time error q times that of the SYN machine.

Although UNSYBs have advantages over SYNs, UNSYBs do not compare so favorably with UNSYs. If we use a timeout mechanism, it must be because for the given application it is critical to always respond within a bounded period T_{max}; there is no other use for the timeout. Therefore, we will say that UNSY fails (not from synchronization failure, but from not meeting other system constraints) if it does not respond by T_{max}. An UNSYB has a timer that it can set it to wake itself up early enough to guarantee that by T_{max} it has already responded.

Let $s = T_{max} - T_{out}$. From equation (2.3), the probability that a given clock cycle will result in a metastability event lasting at most a time t is $f_d W e^{-\lambda_m t}$. The corresponding density function is $D(t) = \lambda_m f_d W e^{-\lambda_m t}$. The failure rate for UNSYs will be given by $\int_{T_{out}+s}^{\infty} D(t)dt$, while the failure rate for UNSYB will be proportional to $\int_{T_{out}-W/2}^{T_{out}+W/2} D(t)dt$. Since $D(t)$ is a very rapidly decreasing exponential curve, the first integral can easily be smaller than the second, as can be seen in the following figure:
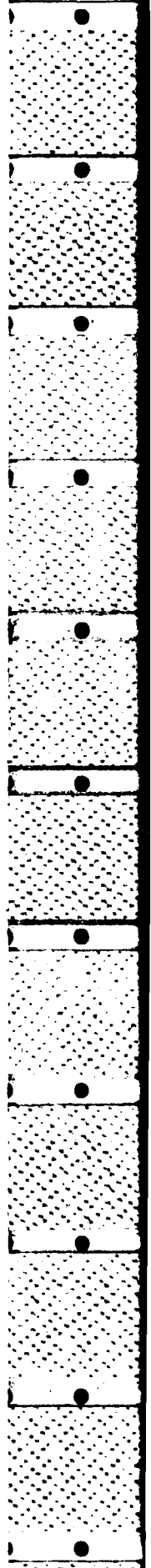
advantages over them.

4.6.5.1 A Synchronizer with a Feedback Timer

Suppose we add to a basic value-safe circuit a timeout mechanism that limits the length of time that a synchronization may take [43]. If after a time T_{out} the signal being synchronized is still metastable, the timeout mechanism will attempt to force the value being resolved into a pre-determined value (*e.g.*, False):
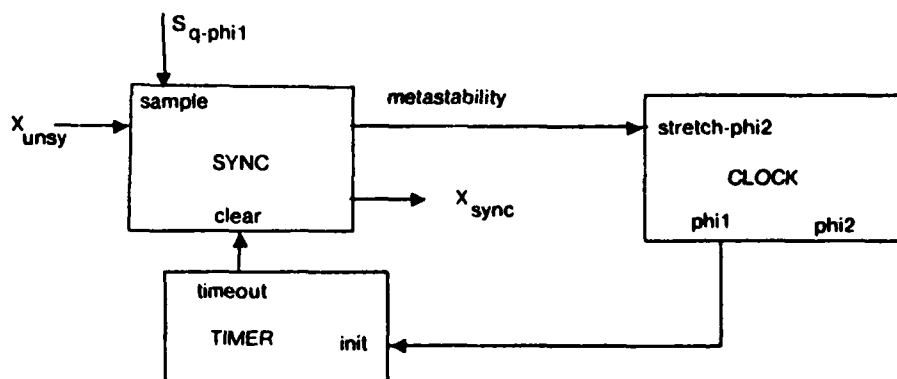
Figure 16. An unsynchronous machine with bounded stretching

The synchronizer has a metastability detector and also an override input that forces the sampled value to zero. The timer is a watchdog that expects to receive input pulses regularly from the clock. If it doesn't receive a pulse for a certain time, say several clock periods, it produces an output pulse. This pulse is used to terminate the metastable state by forcing the sampled value to zero.

From an external viewpoint, the UNSYB has similar properties to those of unsynchronous machines with unbounded stretching (UNSY), except that the UNSYB is no longer absolutely safe, and its worst case time error is bounded. In spite of no longer having absolute reliability, it may have a good reliability. Furthermore, note that the reliability point at which a system works need not be fixed at the time the hardware is being designed, but can be selected by setting the length of the timeout while the system is actually running.

	Synchronous	Unsynchronous
MTBF	$\frac{e^{\lambda_m t_r}}{f_d f_n W}$	∞
Expected Throughput	1	$\frac{1}{1 + \frac{f_d f_n W}{\lambda_m} e^{-\lambda_m t_r}}$
Worst Case Throughput	1	0
Expected Time Error	$\frac{1}{2f_n}$	$\frac{1}{2f_n} + \frac{f_d W e^{-\lambda_m t_r}}{2\lambda_m}$
Worst Case Time Error	$\frac{1}{f_n}$	∞

Figure 15. Performance and Reliability

4.6.5 Unsynchronous Systems with Bounded Stretching (UNSYB)

There are a variety of ideas that make it "virtually impossible" for the system to fail, but still give a bounded time error. Here is one such system. The practical merits of it are not clear, but it will allow us to analyze a typical improvement. We will see that such improvements are not worthwhile because the resulting structures are more complicated than unsynchronous machines, but do not have clear speed or reliability

absolute error will be. Since whenever a machine needs to know the time it can ask some high precision source, the error with which the machine knows the time need not accumulate longer than a clock cycle. Therefore the expected error in the measure of time will be given by the granularity of the clock: $E[\![\text{time error}]\!] = E_{lgth}[\![clk]\!]/2$.

For a fixed-clock system, the expected error is $\frac{1}{2f_n}$. For systems with stretchable clocks, it is:

$$E[\![err]\!] = \frac{1}{2f_n} + \frac{f_d W e^{-\lambda_m t_r}}{2\lambda_m}.\tag{4.5}$$

from equation (4.3). Both synchronous and unsynchronous machines can ask what the time is whenever they need it, so the total error need not be bigger than the error acquired in one clock cycle. Note that although the worst-case time error that can accumulate in a single clock cycle is unbounded for unsynchronous machines, the expected time error can be made virtually as small as that of synchronous machines.

4.6.4 A Summary of Performance and Reliability Measures

In the previous sections we have calculated a number of basic measures of performance and reliability for synchronous and unsynchronous machines. The following table summarizes those results. These results will be used to derive strong conclusions about the relative merits of both kinds of machines when we discuss speed loss, limiting speed, and real-time applications.

where $E_{lgth}[\![clk]\!]$ is the average length of a clock, and $str > t_r$ indicates an event where the clock stretches longer than t_r. Since the clock stretches as long as metastability persists, $P(str > t_r) = P_{met}(t > t_r)$. Clearly $E_{lgth}[\![clk|str \leq t_r]\!] = \frac{1}{f_n}$, since that is the minimum clock period and no stretching beyond t_r occurs. From equation (2.3), $P(str > t_r) = f_d W e^{-\lambda_m t}$.

The remaining factor we still do not know in equation (4.1) is $E_{lgth}[\![clk|str > tr]\!]$, the expected length of a clock cycle when a metastability event is not resolved within t_r:

$$
\begin{aligned}
E_{lgth}[\![clk|str > tr]\!] &= \frac{1}{f_n} + \int_{t_r}^{\infty} t\, P(str > t|str > t_r)dt \\
&= \frac{1}{f_n} + \int_{t_r}^{\infty} t\, e^{-\lambda_m(t - t_r)}dt \\
&= \frac{1}{f_n} + \frac{1}{\lambda_m} \cdot
\end{aligned}
\tag{4.2}
$$

We could also arrive at equation 4.2 by noticing that if we stretch the regeneration period, we are stretching the whole clock cycle by the same amount. For an exponential distribution, the residual probability [20] is also exponentially distributed with the same parameter. Since the expected value of an exponentially distributed random variable is $1/\lambda_m$, the residual probability that gets added to the normal clock length $\frac{1}{f_n}$ is $1/\lambda_m$.

Using equation (4.2) in equation (4.1):

$$
\begin{aligned}
E_{lgth}[\![clk]\!] &= (\frac{1}{f_n} + \frac{1}{\lambda_m})f_d W e^{-\lambda_m t_r} + \frac{1}{f_n}(1 - f_d W e^{-\lambda_m t_r}) \\
&= \frac{1}{f_n} + \frac{f_d W e^{-\lambda_m t_r}}{\lambda_m} \cdot
\end{aligned}
\tag{4.3}
$$

This expression gives the expected frequency f_c, and solving for the throughput yields:

$$
T_{unsy} = \frac{1}{1 + \frac{f_d f_n W}{\lambda_m}e^{-\lambda_m t_r}} \cdot
\tag{4.4}
$$

4.6.3 Expected Time Error

Although the worst case time error can be quite high, we know that the likelihood of such a high error is small, so it is interesting to know precisely what the average

that a flip-flop will remain metastable for a time t or longer is $P(met_t) = f_d W e^{-\lambda_m t}$, where f_d is the rate of asynchronous data transitions and λ_m and W are technology-dependent parameters. The mean time between synchronization failures for a synchronous machine is $MTBF_{syn} = e^{\lambda_m t_r}/f_c f_d W$, where f_c is the clock frequency of the machine.

With stretchable clocks, reaching the end of a clock cycle with the synchronizer still metastable implies that the cycle stretches, but not that the system fails. Therefore, since there are no synchronization failures, $MTBF_{unsy} = \infty$.

4.6.2 Performance: Expected Throughput

Unsynchronous systems are more reliable. What do they lose? Performance: some clock cycles will be stretched, slowing the machine. Next we will show that this loss is negligible. In fact, we will show that we can compensate for this stretching, and actually run *faster* than with conventional machines.

Performance loss comes in two forms: reduced average speed or throughput, and unbounded response time. The bulk of this analysis deals with the average case; we will analyze the worst case when we deal with real-time systems.

Let f_n be the nominal clock frequency and f_c the actual average frequency at which the system runs *taking into account the occasional stretching of clock cycles*. Define the normalized throughput as $T = f_c/f_n$. For a synchronous machine, $T_{syn} = 1$, but for an unsynchronous machine, as more cycles stretch, T_{unsy} becomes smaller. To calculate T_{unsy} we have to take into account that some clock cycles will not result in metastability; some will, but will resolve within a normal clock length; and finally, some will stretch a clock phase beyond its normal length. Solving the conditional expectations, yields the expected clock length and the throughput.

Let $E_{lgth}[\![x|y]\!]$ be the expected duration of an event x, given that a condition y is satisfied. Since the resolution time is not reduced beyond t_r even if the synchronizer settles sooner than t_r, we must split the calculation of $E_{lgth}[\![clk]\!]$ into two parts:

$$E_{lgth}[\![clk]\!] = E_{lgth}[\![clk|str > t_r]\!]P(str > t_r) + E_{lgth}[\![clk|str \le t_r]\!]P(str \le t_r)$$

$$= E_{lgth}[\![clk|str > t_r]\!]P(str > t_r) + E_{lgth}[\![clk|str \le t_r]\!](1 - P(str > t_r)), \quad (4.1)$$

to be met by unsynchronous machines, guaranteeing the value-safe operation of the machine as a whole, given the correct operation of the LMs. In the appendices we provide such a rules system, assuming strict-two-phase LMs. The strict-two-phase assumption is convenient, though not necessary: we can accommodate other organizing principles for the synchronous components by making minor changes in the rules. The rules for designing value-safe unsynchronous systems are accordingly quite general.

§4.6 Quantitative Evaluation of Unsynchronous Machines

In previous sections we have reviewed how metastability can be a problem for conventional synchronous systems and how unsynchronous systems can overcome this problem. To decide whether the solution is good we will compare their performance and reliability with that of conventional machines with non-stretchable clocks. We will show that unsynchronous machines can be clocked faster and are much more reliable than conventional synchronous machines.

This section presents a quantitative analysis of the tradeoffs between time and value uncertainty. As extreme cases, we will obtain the time uncertainty of value-safe systems and the synchronization failure rate of conventional systems. We will also explore systems with "bounded stretching" of the clock; such systems have properties that are midway between those of fixed-clock machines and those of unsynchronous machines. These models yield worst-case and average-case measures of performance and reliability. The MTBFs and a new normalized throughput measure prove that unsynchronous machines are superior to conventional synchronous ones for many high-performance applications, including some real-time applications.

4.6.1 Reliability

To compare quantitatively the effects of metastability on conventional and unsynchronous machines, we need reasonable models for the behavior of regenerative elements (*e.g.*, flip-flops) and of the communicating subsystems.[2] From Chapter 2, the probability

---

[2] Hammurabi discussed some reliability issues: "If a house fell and his owner is dead the builder is to be dead," but he seems not to have cared about synchronization failures.

In the figure above, system A receives X_{unsy}, which it samples during φ_1. It never needs to stretch φ_2, but occasionally it may have to stretch φ_1. It produces an output Z that is synchronous for A, but since B runs from an independent clock, it must go through a synchronizer before B can use it. B uses some other external signal Y, which is also synchronized. Since B samples both Y and Z during φ_1, it may have to stretch φ_2 if any of B1 or B2 go metastable. Note that if only one of B1 or B2 is metastable, the other one keeps the value it has sampled safely until *both* are ready to go on. Note that if an LM had for some reason sampled data both on φ_1 and on φ_2, the corresponding clock would have had stretch inputs for both phases.

Since the unsynchronized signals have no semantics associated with them, the programs running in each machine will determine their meaning. For example, Z might be a request for some resource, and X its corresponding acknowledge, while Y might be the output of an A/D converter, providing the temperature of an engine. In such a case, the X/Z pair might follow some particular protocol, while clearly Y would not. It might be that A is the master and B a slave or vice versa, or perhaps B is an arbiter and Z/Y are requests for a resource. Note that value safety does not imply "correctness". For example, if A sends values to B much faster than B can read them, B will lose values. Hence, appropriate protocols are necessary. In any event, the system remains value-safe.

§4.5 Automatic Verification

We have discussed a way of building global structures that link together many locally synchronous machines. Assuming the correct operation of the synchronous components, it would be useful to provide guarantees about the operation of the global unsynchronous machine as a whole.

In a sense, unsynchronous machines are fairly simple, since the purely synchronous part of each LM remains unaltered, and, assuming the correctness of each component LM, we only need to make sure that every external unsynchronized signal is properly synchronized. Verification can be restricted to the interfaces between LMs because the modularity of the LMs is preserved by the shielding synchronizers. It is only necessary to check that the unsynchronized signals are synchronized to the appropriate clock phase and that all the stretch signals generated by the synchronizers are appropriately connected to the stretch inputs of the clock.

Note that rules can provide an efficient way of stating the constraints that have
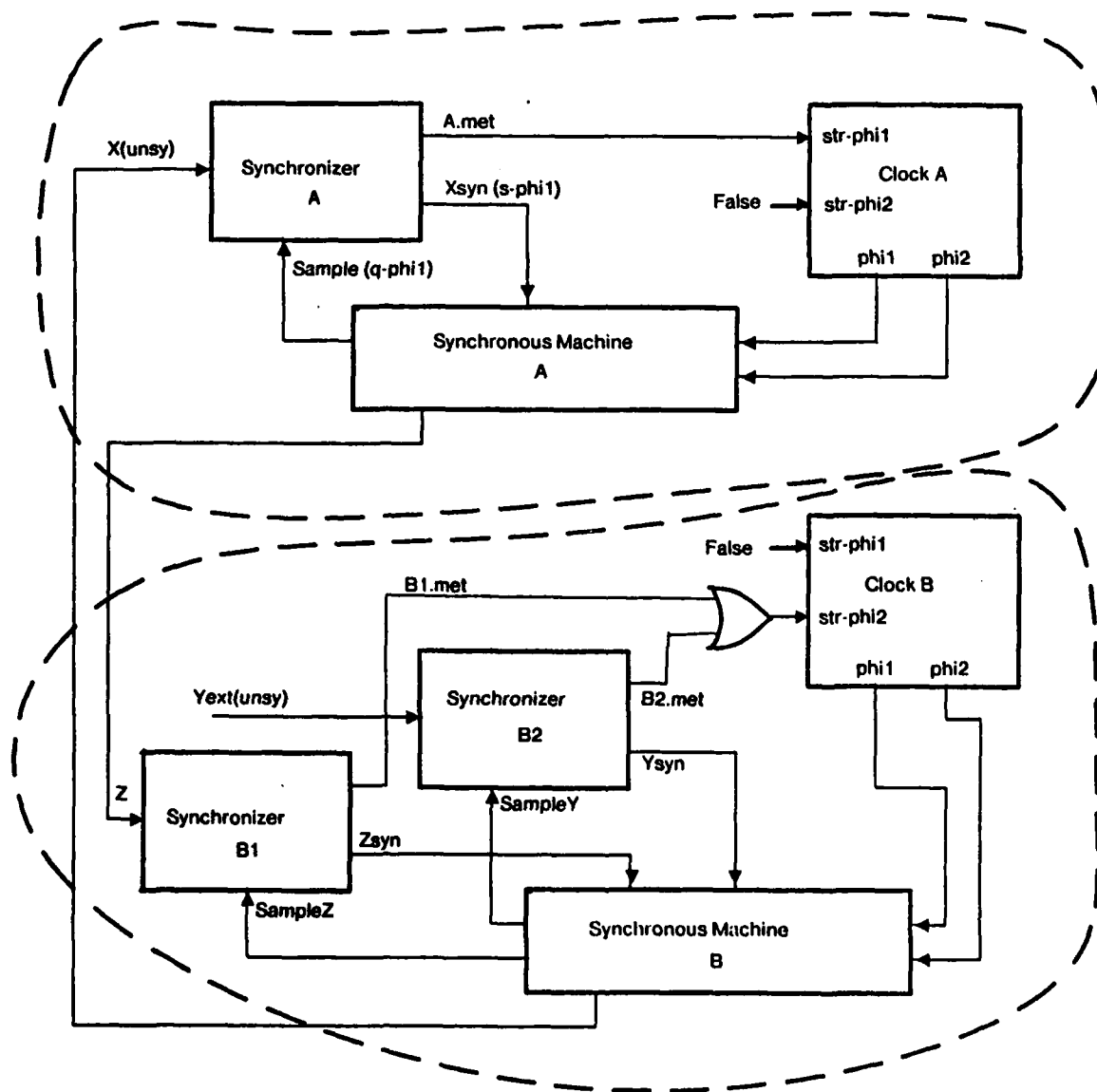
Figure 14. A Basic Value-Safe Unsynchronous System

§4.4 An Unsynchronous Machine with Two-Phase Clocking

For the sake of concreteness we will show next an unsynchronous machine composed of several locally-synchronous machines (LMs). Each of the LMs can be a synchronous, strict-two-phase machine encapsulated in a shell that provides a clock and an interface with other LMs. Here we use the clock and synchronizer that was described above.

Each LM must use the clocking signals provided by its own stretchable clock. All asynchronous inputs to each LM are synchronized with SMDs, and the corresponding stretch signals are ORed together and fed to the stretch input of the local clock. In an appendix we provide a formalization of this structure, which consists of rules that allow us to verify the value-safety of an unsynchronous machine. What follows is a simple example that shows how we can tie the components together.

In the unsynchronous system in the next figure, we have two LMs exchanging unsynchronized signals. They also receive unsynchronized signals from the outside. Each unsynchronized signal is fed to a synchronizer, which transforms it into a $stable\varphi_1$ signal. The OR of the metastability detection lines coming out from all the synchronizers in each LM is sent to the $stretch_{\varphi_t}$ input of the local stretchable clock.

If a stretch-φ_2 input is raised as a v-φ_2 signal, nothing happens to φ_2 if stretch-$\varphi2$ drops before $\varphi_2^{\downarrow}$. On the other hand, if stretch-$\varphi2$ remains high after the time when the corresponding PG goes low, it will prevent φ_2 from falling, and will stretch it for as long as the stretch remains asserted. Meanwhile, φ_1 is not being retriggered, but once φ_2 goes low, a φ_1 pulse will be emitted, and the clock will proceed normally. Notice that a race in between the falling of PG's output and the falling of a stretch line produces no glitches whatsoever because they are being ORed, so the output will consist simply of the one falling last.

(d) A synchronizer with metastability detection (SMD): A φ_1-clocked D flip-flop samples the input data, and feeds its Q and $\overline{Q}$ outputs to an analog comparator whose output will be low only if Q and $\overline{Q}$ differ by more than a given threshold. The output of the comparator is ANDed with a φ_2 clock signal.
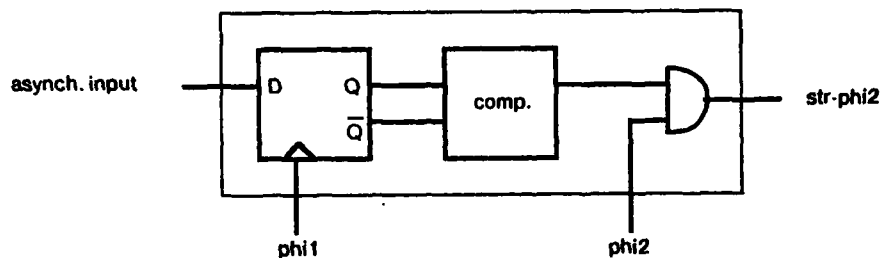
Figure 13. A synchronizer with metastability detection

The SMD samples data on φ_1 only, and the only way in which its flip-flop can go metastable is if X is changing at $\varphi_1^{\uparrow}$. In this case, Q and $\overline{Q}$ will be similar,[1] instead of having complementary values. Hence, if the metastability does not subside until $\varphi_2^{\uparrow}$, stretch-φ_2 will be asserted as a v-φ_2 signal. If metastability hasn't subsided by the time the PG corresponding to φ_2 goes low, φ_2 will be stretched as long as necessary. Note that we can set the comparator so that it begins to drop its output when Q and $\overline{Q}$ exit from the digitally-undefined region. When the flip-flop eventually stabilizes, the comparator will make stretch-φ_2 fall, so we can latch Q with $\varphi_2^{\downarrow}$ with the certainty of latching a digitally-defined value, and the resulting machine is value-safe.

---

[1] Though not true of every flip-flop, over-damped symmetric flip-flops have this property. Once their outputs differ by more than a given value, it is guaranteed that within a bounded interval of time they will have stabilized completely to digital values.

$$t_r = d/f_n.$$
(4.9)

From equation (4.4), the throughput of an unsynchronous machine is:

$$T = \cfrac{1}{1 + \frac{f_d f_n W}{\lambda_m} e^{-\lambda_m t_r}}.$$
(4.10)

Since $T = f_c/f_n$,

$$\frac{f_c}{f_n} = 1 - \frac{f_d f_c W}{\lambda_m} e^{-\lambda_m t_r},$$
(4.11)

and we can calculate the limiting speed by taking the limit of f_c when f_n tends to infinity:

$$\begin{aligned}
f_{c\,\text{max}} &= \lim_{f_n \to \infty} \cfrac{f_n}{1 + \frac{f_d f_n W}{\lambda_m} e^{-\lambda_m t_r}} \\
&= \lim_{f_n \to \infty} \cfrac{1}{\frac{1}{f_n} + \frac{f_d W}{\lambda_m} e^{-\lambda_m d/f_n}} \\
&= \frac{\lambda_m}{f_d W}.
\end{aligned}$$
(4.12)

Hence

$$f_{c\,\text{max}} f_{d\,\text{max}} = \frac{\lambda_m}{W}.$$
(4.13)

This equation tells us that for unsynchronous machines, the product of the effective clock frequency and the frequency of asynchronous perturbation have a technology-dependent limit. If we attempt to drive an unsynchronous system too fast, eventually the system will self-adjust to a limiting speed controlled by the decay speed of its flip-flops, and not by f_n. Therefore, it is important to know if this limit is high enough.

For typical values for 4μ nMOS technology, we get that $f_{c\,\text{max}} f_{d\,\text{max}} \approx 5\ 10^{14} \text{HZ}^2$, which is so high for this technology that it does not limit us in any way. In general, for any technology, we will reach other limits well before reaching anything close to the unsynchronous limit speed.

4.6.8 Real-Time Systems

For those systems for which we are concerned only abou. the reliability and average performance of the system, delayed results are clearly better than timely garbage. However, there are some real-time systems for which the response time is critical. The key issue is that for different people "real-time" means completely different things: a banker needs real-time processing of transactions; a steel mill needs real-time control of the roller position to produce a sheet of uniform thickness; a plane has real-time control of its turbines; a dynamic memory has to be refreshed in real-time, etc. To differentiate among all these, define a critical time t_c as the maximum time allowable for synchronization.

Except when we discussed unsynchronous machines with bounded stretching, we only considered unsynchronous machines with absolute reliability. Now we have to contend quantitatively with the possibility of failure due to slow response. Using equation (2.3) gives the expected time until a metastability event lasting t_c, from which we derive the reliability of an unsynchronous machine for a given real-time application:

$$\text{MTBF}_{unsy} = P(str > t_c) = \frac{e^{\lambda_m t_c}}{f_c f_d W}. \tag{4.14}$$

To give a feeling for the numbers we are dealing with, for $t_c = 1msec$, MTBF_{unsy} for 4μ nMOS is several times the estimated life of the Earth. This result is rather encouraging, but not conclusive yet, because exponentials shrink as fast as they grow. Under tighter conditions we would have to re-check our numbers. However, we can instead obtain a technology-independent quality measure (Q) that will allow us to decide in a more general way whether an unsynchronous structure is appropriate.

Let $Q = \text{MTBF} f_{max}$, where f_{max} is the maximum clock frequency at which the system runs, and the MTBF takes into account both synchronization failures and failures to respond in time. MTBF_{syn} will be given by equation (2.4), and MTBF_{unsy} will be given by equation (4.14), so

$$\frac{Q_{unsy}}{Q_{syn}} = e^{\lambda_m(t_c - t_r)}, \tag{4.15}$$

where t_r is the regeneration time allotted for the synchronous machine.

In some applications (*e.g.*, data capture) a synchronous system can pipeline the data so as to have a longer synchronization time, making $t_r \approx t_c$. However, at $t_r = t_c$,

the synchronous system is always at the limit of failing to respond quickly enough. Therefore, the designer must build in a safety margin, so t_r will be smaller than t_c. Let $k_s < 1$ be the safety factor, where $t_r = k_s t_c$. For any technology, the regeneration time t_r should be several times larger than the expected decay time of a flip flop. Therefore $t_r \gg 1/\lambda_m$, and

$$\frac{Q_{unsy}}{Q_{syn}} > e^{1/k_s - 1}. \qquad (4.16)$$

Hence, if pipelining is possible, Q_{syn} can be almost as good as Q_{unsy}.

However, there are applications for which both average and worst case response times are important (*e.g.*, arbitration), where pipelining may not be applicable. An unsynchronous machine can do up to one arbitration per clock cycle, and it fails when it does not respond within t_c. A synchronous machine doing the same task fails when it does not resolve an arbitration within $t_r = 1/f_n$. As expected, if because of real-time constraints, the system fails as soon as the clock is actually stretched at all ($t_c = 1/f_n$), the unsynchronous machine must run at the same speed as the synchronous one to have the same reliability. Replacing $1/k_s$ with $t_c/t_r = t_c f_n$ in equation 4.16, we get:

$$\frac{Q_{unsy}}{Q_{syn}} > e^{t_c f_n - 1}. \qquad (4.17)$$

However, $t_c > 1/f_n$ for any reasonable design, because otherwise the systems would have no safety margins. By allowing t_c to be just a few clock cycles, the unsynchronous machine will arbitrate with a *speed.reliability product* hundreds of times higher than a synchronous machine built in the same technology.

Since reasonable real-time constraints cannot be at the granularity level of the clock, we can conclude that even for real-time applications unsynchronous systems are better than synchronous ones.

4.6.9 High-Precision Stretchable Clocks

The low probability of long stretchings opens another design possibility: if for some reason we needed   a stretchable clock that kept time with the precision of a crystal clock, we could phase-lock it to a crystal clock, using an extremely slow feedback locking loop. Stretching a cycle would not interfere with the locking mechanism, which would not respond quick enough to alter the basic clock frequency due to a stretch event, but

nonetheless would make it run in-phase in the long run. The only draw-backs would be that to start up the clock it would take many more cycles than for a normal crystal clock, and also that with some non-zero probability the locking could interfere with the stretching if the stretching lasted for a period that were comparable to the time constants involved in the feedback loop.

Alternatively, as suggested in [Hann], two variable speed clocks layed on the same chip can be controlled by the same voltage reference, so their normal frequency is quite close. One of the stretchable clocks is phased-locked to a crystal reference, and its phases are never stretched. The other stretchable clock, which produces the actual clock signal used by the system, can have its phases stretched. In this way, the stretching does not interfere with the phase-locking, independent of the stretch duration. The normal frequency at which the system runs is not exactly that of the crystal reference, but will be quite close. For example, the processor in the next figure uses this scheme to obtain a stable frequency from a stretchable clock [15]. (The cyclic shift register divides the locking-frequency so that a lower reference-frequency can be used [14].)
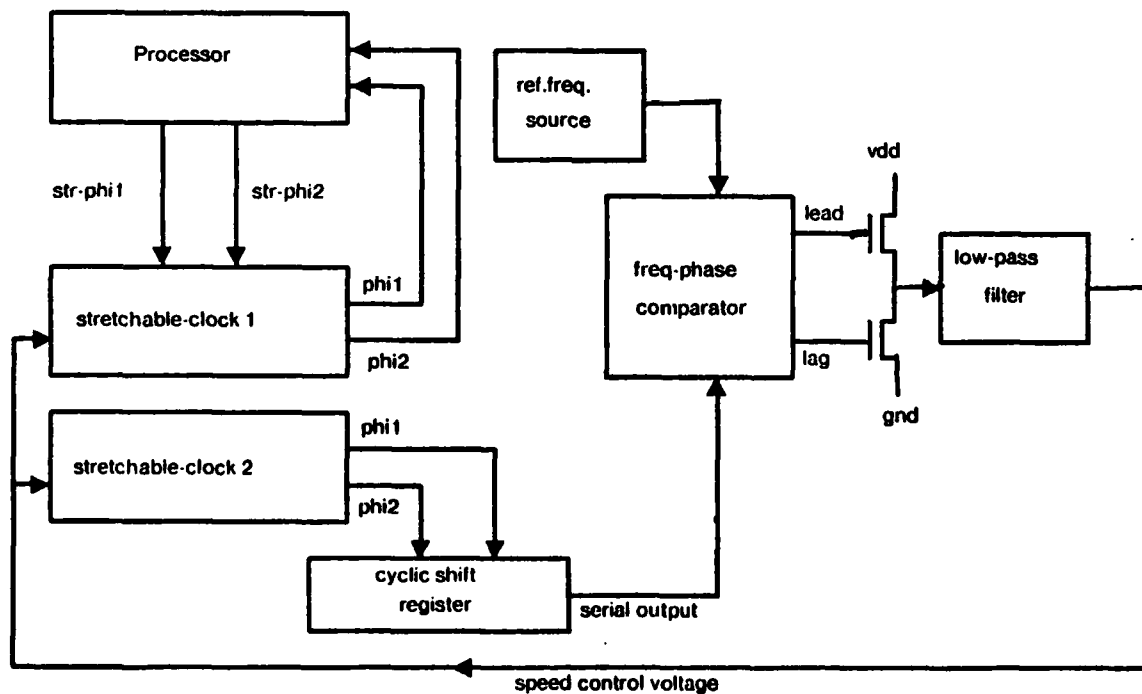
Figure 20. Phase-Locking a Stretchable Clock

§4.7 Conclusion

We have seen a very general organization for building value-safe unsynchronous machines, and we studied the relation between the global unsynchronous structure and a particular organization for the local synchronous machines. We analyzed quantitatively the trade-offs between reliability, throughput, and real-time constraints, and concluded that unsynchronous machines can run at much higher speeds and with much higher reliabilities than equivalent synchronous machines.

Escapement Systems

When we dealt with unsynchronous systems we did not use any information about how or when asynchronous inputs could change. In contrast, an escapement organization (EO) does not know when an external asynchronous signal may change, but knows in advance the direction of the transition. We will use this knowledge about the asynchronous signals to build value-safe GA-LS systems without synchronizers of any kind.

From the communications point of view, unsynchronous systems are very general, but if we do not need such generality, their generality wastes time and area. By using more structured signals that obey certain protocols, EOs have the following advantages over unsynchronous systems: less area, since EOs do not use a synchronizer; faster response, since EOs do not need synchronization clock cycles; and deterministic response time, since EOs have no metastability detector to wait for. Lastly, their clock re-starts in phase with incoming external asynchronous signals.

In Section 1 we describe the escapement mechanism, which is used in Section 2 to assemble value-safe GA-LS machines. Section 3 discusses some optimizations that improve the performance and reduce the area needed to implement an EO. Next, Section 4 analyzes how to provide correctness guarantees through verification of the EO specifications and compilation of the specifications onto hardware. Finally, Section 5 evaluates the performance of EOs.

§5.1 The Escapement Mechanism

To develop a feeling for the operation of EOs, we show the operation of a simple escapement machine (SEM) that embodies the second "fundamental solution" approach suggested by Pechoucek in [37]. The EO mechanism must know in advance the direction of each transition that the external signal will make. To build a complete EO requires some kind of communications protocol. For convenience, we will use the 2-cycle communication protocols described in Chapter 2.

We trace the flow of control throughout a transaction of this SEM with an external slave, as shown in the figure below. Initially both Req and Ack are low while the SEM is computing. When Req is raised the clock of the SEM is stopped until Ack is received. When the SEM restarts it knows Ack has arrived (otherwise it would still be stretching φ_2), so it can do other computations and eventually it can lower Req, again stopping the clock. Eventually, Ack will fall too, bringing the system back to its initial state. Note that the SEM receives asynchronous acknowledges without using a synchronizer.

Block Diagram   Logic Diagram
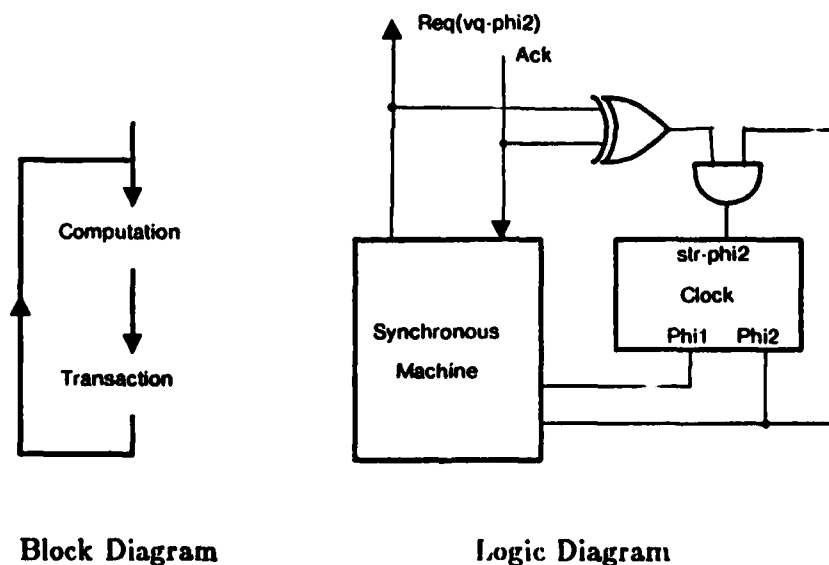
Figure 21. A Simple EO Master

In terms of a state diagram, suppose that after state S1 (see next figure) the EO is ready to receive some asynchronous request, which will be handled in S2. We denote such conditional transitions by changing the normal transition arcs used in state diagrams to squiggles labeled with the transition condition.

S1 (ready for Req)

← Req

S2 (handle Req)
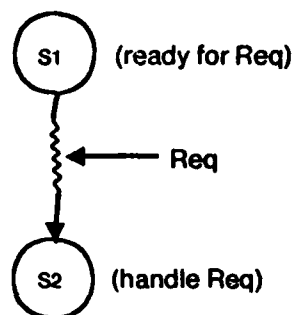
Figure 22. Escapement Stretching: State Diagram

The squiggle can be interpreted as stretching state S1 until the external condition Req is satisfied, as in the figure below:

Req
S1

stretch

Clock
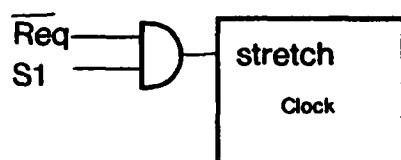
Figure 23. Escapement Stretching: Implementation

The behavior of this system can be readily seen in the following timing diagram. Note that as soon as soon as the request arrives the system can enter S2 and start handling it, without ever having actually sampled the request signal.

state   S1   S2

Req
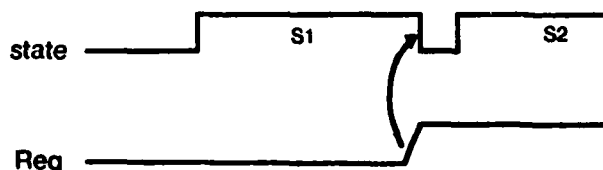
Figure 24. Escapement Stretching: Timing

The escapement mechanism does not have any hidden metastability problem. Examining unsynchronous machines gives insight about how EOs avoid problems with metastability. When an unsynchronous machine receives an external unsynchronous signal, it uses this signal as a discriminant to choose among different states into which it can make a transition. Using the squiggle notation loosely, note that unsynchronous machines stretch a clock phase until metastability subsides, and then make a transition

to one of two states (corresponding to a Boolean decision), as shown in the next figure:
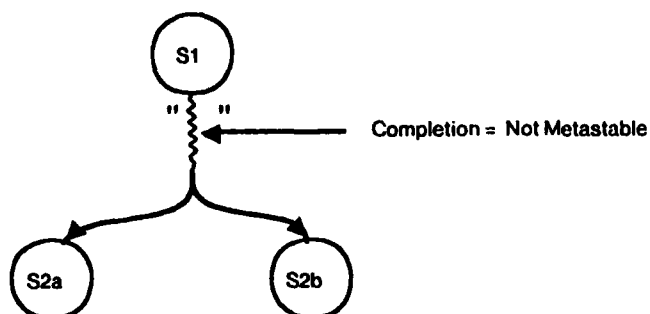
Figure 25. Stretching in Unsynchronous Systems

Unsynchronous machines circumvent the $\mathcal{DRV}$ theorem by giving unbounded time for a decision. EOs avoid such decisions altogether: they have only non-bifurcating squiggles, as shown in the next figure, and cannot choose between different states based on an external asynchronous signal
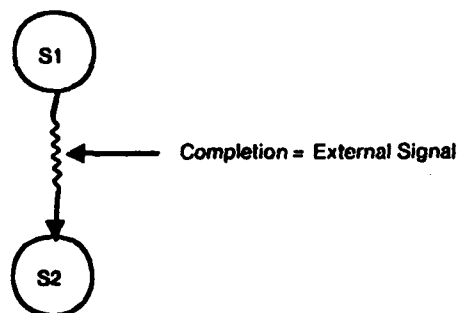
Figure 26. Stretching in Escapement Systems

The escapement "wait" is also different from a conventional synchronous busy wait (which is not value-safe):
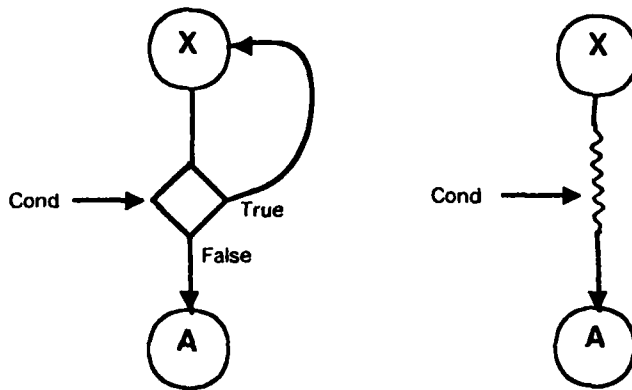
Figure 27. Difference Between Busy Wait and Stretch Wait

Having seen the differences between the ways in which escapement, unsynchronous, and synchronous systems handle asynchronous inputs, we can focus on how the escapement mechanism avoids metastability problems. Remember from Chapter 4 that for the clock to operate correctly the stretch-φ_2 input to the clock must be asserted as a rq-φ_2 signal, and that the only constraint on the falling edge is that it cannot rise again in the same clock cycle once it falls. Call $S_0, S_1, S_2, \ldots$ the sequence of states traversed by the EO. If a request must arrive before the EO enters state S_1, the request will not go away before the EO emits an acknowledge, and the request must fall before the EO enters state S_2, we can make stretch$\varphi_2 = \varphi_2 \wedge (S_0 \wedge \overline{Req} \vee S_1 \wedge Req)$.

Consider what may happen when $Req^\uparrow$. First, when $\varphi_2^\uparrow$ in S_0, Req may still be zero, in which case the EO stretches from $\varphi_2^\downarrow$ until the request arrives. Second, the request may have already arrived, in which case no stretching at all occurs, and the EO proceeds to S_1. Finally, the request may arrive concurrently with $\varphi_2^\uparrow$, in which case the stretch-φ_2 line will glitch for a bounded period after $\varphi_2^\downarrow$ without any effect on the clock. Note in all three cases that once stretch-φ_2 falls, it will not rise again within S_0. The behavior going from S_1 to S_2 when $Req^\downarrow$ is analogous with the exception that the stretching occurs while Req is high. Since the EO handles all asynchronous inputs in this way, these inputs cannot trigger metastable conditions anywhere else, and as long as both communicating machines respect the communication protocol they will remain value-safe.

There is an interesting parallel with the unsynchronous mechanism: the stretch signals that are generated by the escapement mechanism interact with the clock in exactly the same way as the stretch signals generated by the metastability detector of

unsynchronous machines. Hence, the synchronizer with metastability detector can be considered as an interface of an escapement machine that allows it to receive unsynchronized signals.

There is also an important difference with the unsynchronous mechanism: while an unsynchronous machine can sample numerous lines, one after the other, until it finds some condition it is looking for, an EO will stop as soon as a condition to traverse a squiggle-arc is not satisfied. Hence, unsynchronous machines can poll multiple lines effectively, but EOs cannot.

§5.2 Structures for Escapement Machines

This section explores the kind of structures that can be built using the escapement mechanism. For clarity, some of the chosen examples are not as fast or compact as they could be and are used mainly to emphasize different aspects of EOs. In particular, we will discuss various structures suited for controllers, servers, pipelines, and arrays.

The EOs will be described with logical diagrams and also with state diagrams extended with the squiggle-arc notation (ESDs). Each squiggle will have associated arrows labeled with the conditions required to traverse the squiggle. The source of the arrow will indicate the source of this signal, which typically will be some state in the ESD of another LM.

5.2.1 Basic EOs

We have seen that EOs must communicate using some protocol (*e.g.*, the 2-cycle protocol) to know in advance in which direction the asynchronous inputs may change. The simplest possible state diagrams that implement a 2-cycle protocol are the two interlocked loops shown in the figure below:
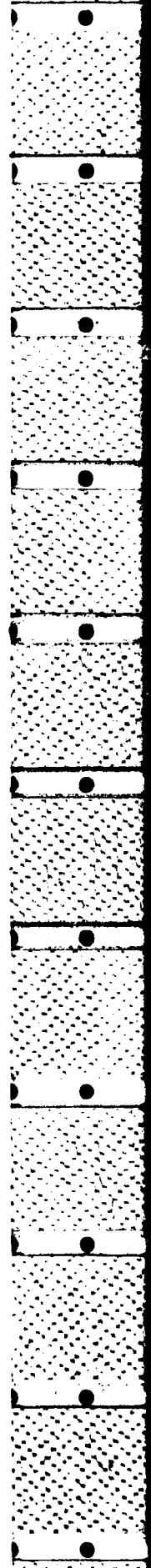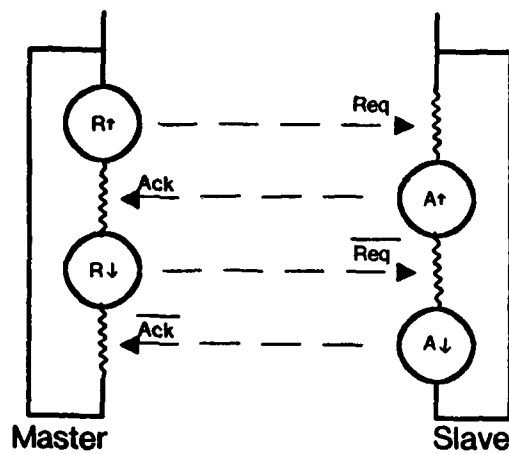
Figure 28. A Basic Master-Slave EO

e diagram above shows only the control flow. Between each one of those control
ions we can insert a computation, which is denoted by a C_i. The C_is are arbitrary
ions that the LM may carry out, including both computations and communication
h other LMs. The ESD in the following figure has all the C_is possible in a basic LM
p.
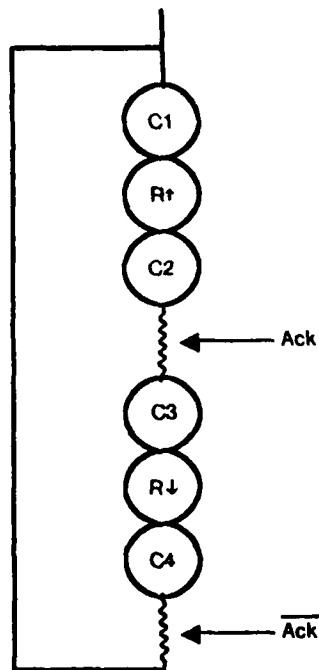
Figure 29. State Diagram of a Basic Master LM

The C_is form the basis for a notation used to distinguish among basic LMs. The achine above is a C1234 EO. It can support a 2-cycle protocol. Eliminating C3 and C4 ·oduces a C12 EO, which can not do more than one data transfer per full handshake, only a 4-cycle protocol could be supported (see Chapter 2 for a brief discussion on ·otocols).

2.2 Mapping the Extended State Diagrams onto Hardware

The ESD notation is unambiguous and complete for EOs, and the transformation ESDs into logic diagrams is straight-forward. The following two rules indicate how do this mapping:

● Multiple Arrows: If there are n arrows with associated transition conditions $Cond_i$ pointing to the same squiggle, they are transformed into a single arrow with an associated condition $COND = Cond_1 \wedge Cond_2 \wedge \cdots \wedge Cond_n$.

iggles: For each squiggle with an associated transition condition $Cond$, there
be a signal $Last$ that indicates the completion of the task preceding the
ʒle. $Last$ must never be raised at any other point in the ESD. For each
ʒle in the LM, there will be a stretch signal $S = Last \wedge \overline{Cond}$. The OR of all
retch signals is fed to the stretch input of the clock.

ʋ these rules are used to map the Master LM of the figure below onto a logic
. For simplicity assume that $Req^{\uparrow}$ and $Req^{\downarrow}$ are s-φ_2 and that two s-φ_2 signals
the completion of C2 and C4 respectively.
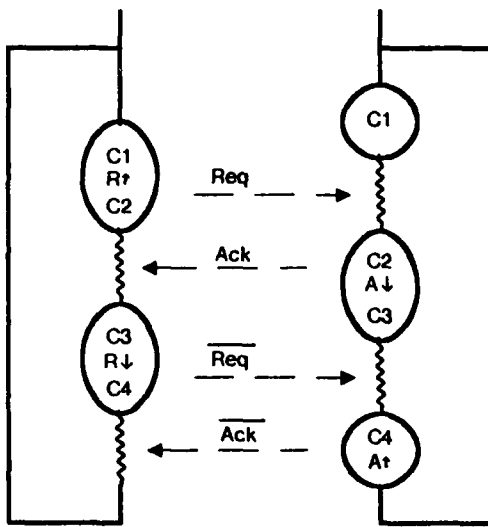
Figure 30. ESD for a Master C1234 EO

ng to the second mapping rule, for each arrow, form the conjunction of the last
ʒceding the squiggle and the external condition for stretching, generating two
signals: $Last(C2) \cdot \overline{Ack}$ and $Last(C4) \cdot Ack$). OR these stretch signals and send
, the clock, as shown in the following figure:

ts its own clock. Since achieving phase lock takes some time, each block is
ceded by a preamble whose only purpose is to ensure that when the actual data
ts, the recipient will sample it in the middle of each frame. For high speeds,
 preamble is many clock cycles long, so if the blocks are short the loss may
significant (Moreover, sometimes metastability will prevent the receiver from
ieving lock in the allotted time).

EO combines the advantages of both schemes but without oversampling or a
amble: The sender starts each block with a start bit. The recipient EO only
ds to stretch until it is awakened by the start bit. Since the EO starts its
k in phase with the incoming data, it has no problem in sampling directly the
Idle of each frame. If blocks are long, the sender must also encode a clock signal
h the data, to which the EO will lock after some number of cycles. Note that
 EO can start sampling data while it is achieving lock because it starts being
cisely in phase, instead of starting with a random phase. Thus the EO does
 lose time during a preamble and does not need to over-sample. Note that this
roach requires that the rising edge of the start bit not have dynamic hazards.

ynthesis of EOs

hough the escapement mechanism by itself is simple, the resulting EOs can be
nplex. This section shows methods to handle the design complexity. Our design
ology uses a specification formalism and a set of rules that control how LMs
mbled and connected to other LMs. The purpose of the specifications and the
two-fold: helping the designer with a vocabulary and a structure that makes
gn process simpler, and also guaranteeing that the resulting machine complies
 specifications.

ification and automatic synthesis are two paths followed to obtain programs
 guaranteed to be correct respect to their specifications. The fundamental
y we want guaranteed is value-safety. Unfortunately, as is shown in an appendix,
fety of EO circuits is undecidable. Hence, it is impossible to create a program
 verify an arbitrary circuit and tell whether or not it is a value-safe EO.

erefore, we developed a low-level specification formalism that can be compiled
into hardware. The class of machines that can be described with this formalism

.4 Performance and Reliability of Escapement Machines

The operation of EOs does not have the interesting probabilistic characteristics of synchronous machines, but in fact that is better: the analysis is exceedingly simple. response time can be bounded with probability 1. It does not need synchronizers to value-safe, and of course cannot suffer synchronization failures since it does not do y synchronization.

The following two characteristics give EOs them their main performance ad- ntages: (a) EOs respond immediately once a handshaking signal arrives, and (b) EOs itart the clock in phase with the edge of the handshaking signal.

⎸ Unsynchronous machines can also handle the protocols that are mandatory for EOs, but for unsynchronous machines, a full handshake involves the reception of 4 unsynchronized signals ($Req^\uparrow$, $Ack^\uparrow$ $Req^\downarrow$ and $Ack^\downarrow$). Each of the four synchroniza- tions takes one clock cycle for the actual synchronization, plus an average of half a cycle that is lost because the signal has to arrive before the actual sampling occurs. Therefore unsynchronous machines spend 6 clock cycles per handshake for synchronization, versus none used by the EO.

) The following example will show the advantages of starting in phase. Suppose a system must receive serial data at extremely high speeds. Each data block starts at arbitrary times, but the bit-rate is known to some error ϵ. There are two basic approaches that are used to receive blocks of data, which are exemplified by UARTS and by the Ethernet, and EOs combine the advantages of both.

1) In a UART, the receiver *over-samples* (typically at 16 times the bit-rate) to detect the rising edge of a *start-bit* (a 1 bit which indicates the beginning of a block). Having detected the edge of the start bit with reasonable accuracy, and knowing the bit-rate, it can sample each bit in the middle of its *frame* (the interval in which a bit value appears on the line). This approach has multiple limitations: the blocks must be short enough that the receiver does not drift too far off from the middle of each frame, the bit-rate cannot be really high, because over-sampling requires the receiver to operate at speeds much higher than the bit-rate; and sampling for the edge of the start-bit can result in a metastable state.

2) In an Ethernet, the data carries an embedded clock signal to which the receiver

LM must emit a SetRequest, and later it must emit a Join signal indicating C2 is
ete. The Join signal is ANDed with the stretching condition $[Req \lor Ack]$. Thus,
gnal is necessary for the fork, and another one for the join. This section shows
, single signal is sufficient when LMs use 2-phase clocking.

ransform the ESD of the parallel C2 master as shown in the next figure. Note
nside the loop the ordering of operations has not changed. In the entry to the
there is now an additional join preceding the first fork. This is of no consequence
se the join condition must hold before executing a fork. For example, for a basic
nly an initialization that clears all handshaking signals can precede the basic loop.
:, the first join precedes the first fork in the transformed EO, but it corresponds
tretch that does not take place.
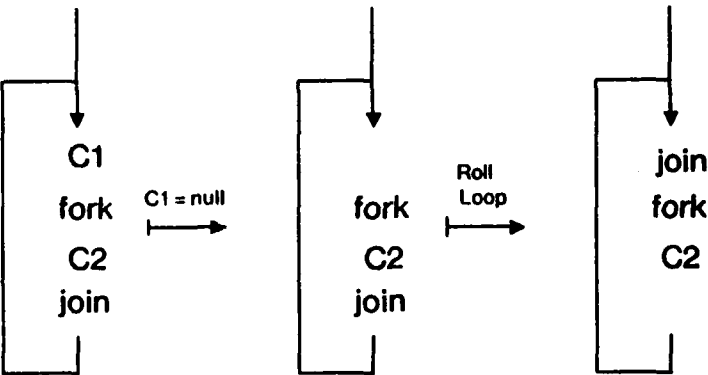
Figure 45. Rolling the C2 Loop

.ct fork-vφ_1 ,sφ_2 be emitted by the synchronous control of the EO. Qualifying the
iignal with φ_1 , we obtain the Join-vqφ_1 signal; qualifying the fork signal with φ_2 ,
itain SetReq-qφ_2 . Hence, we can use a single signal, by displacing the fork and
vithin the basic loop, as in the figure below.

Figure 46. Packing Fork and Join
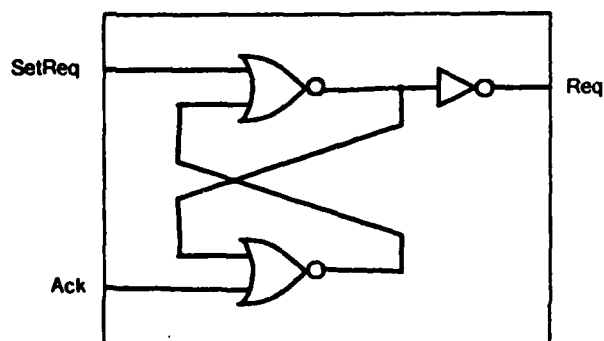
Figure 43. Master-Set SR Flip-Flop

The master-set flip-flop is faster, since it need not delay the out-going request until SetReq is low, and it requires no assumptions about relative delays. Whichever implementation is chosen, we will call this circuit a fork box.

Finally, to take care of the join, a stretch on $Req \lor Ack$, placed at the bottom of the ESD loop, will ensure that the LM can proceed with a new C1 only after the Ack is received. The following figure shows the resulting EO. Note that even if C1 is empty, the optimization still applies.
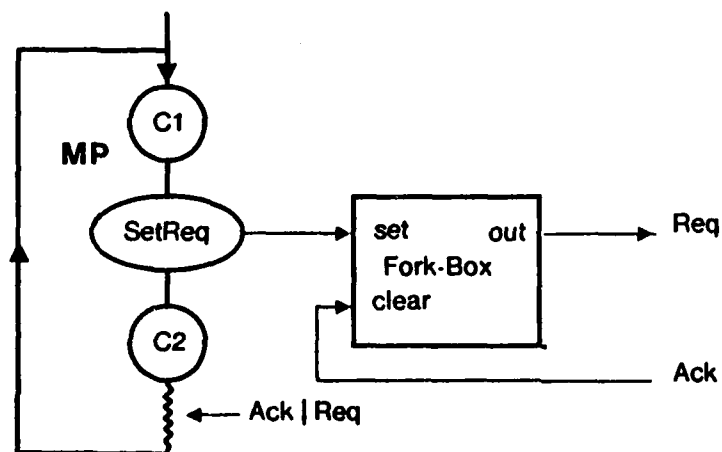
Figure 44. Parallel C2 or C12 EO

5.3.2 Signal-Packing in Parallel C2 EOs

For the parallel C2 master shown in the previous section, the synchronous control

- The only task in the life of FP is to clear Req when Ack comes.

- The purpose of the join is to complete the current cycle of the LM before starting a new one.

Let us modify the C12 EO, but using only valid EO primitives and preserving its ɔgical characteristics. Req can become accessible to MP and FP by storing Req in an ;R flip-flop that can be set by MP's signal SetReq and cleared by FP. Conceivably, ſ the slave is very fast compared with MP, the acknowledge could come back while he MP is still setting the flip-flop, possibly ending in a metastable state. Since the ɪrotocols guarantee that Ack will not arrive before Req is emitted, it is enough to delay ending out Req until SetReq falls (SetReq stays high for a single clock phase), as shown ɑ the next figure:
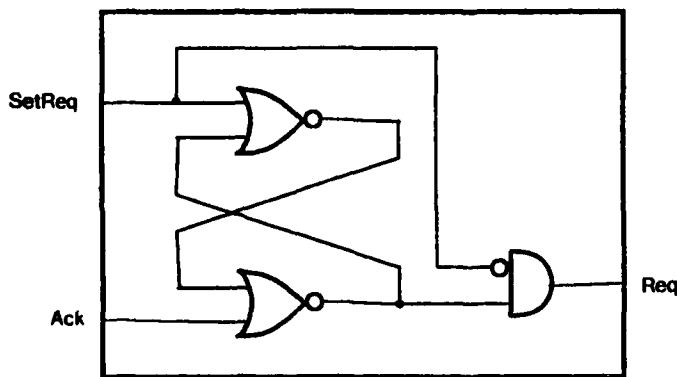
Figure 42. Set-Inhibit SR Flip-Flop

Note that we must ensure that the inhibition path from SetReq to Req is faster than the ɪath from SetReq to Req via the NOR gate; it is easy to implement a circuit for which his delay assumption always holds. Nonetheless, a less obvious and better solution is o use a master-set SR flip-flop:
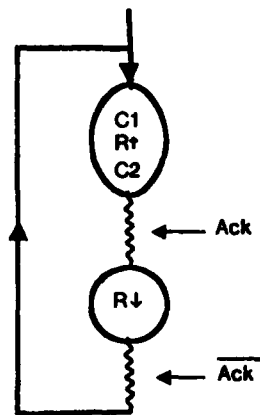
Figure 40. A C12 EO

In the EO shown above, C2 is already running in parallel with the $[Req^\uparrow, Ack^\uparrow]$ half of the communication, but on the $[Req^\downarrow, Ack^\downarrow]$ half, it must idle. Hence, C2 should occur in parallel with the second half of the communication, as shown in the next figure. The EO has been transformed into a main processor (MP) and a fork processor (FP).
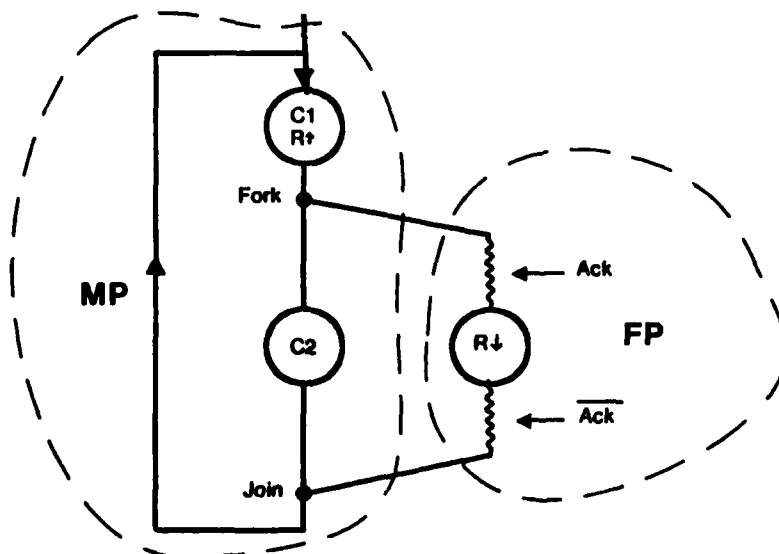
Figure 41. A C12 EO with a Fork

Notice that:

- Neither forks nor joins are available primitives in escapement systems.

- Req is a variable shared by two processors: MP sets it; FP clears it.

would not be able to tell that such a switch of mastership has occurred. Note that this solution is analogous to what we do to switch mastership with unsynchronous machines. The switch of mastership is hidden from the hardware, which becomes just a substrate on top of which a more complex behavior is implemented in "software". Therefore, at the hardware level the master/slave relationship can conveniently remain static (the "hardware master" initiates the communication), while the mastership switch occurs in the next layer [44] above the EO hardware.

§5.3 Optimization of EOs

In basic EOs, while signals propagate between LMs, the LMs are idle. This lost time is significant for C2 and C12 EOs, but it can can saved by overlapping communication and computation. This section shows how EOs can be modified to achieve this overlap while remaining value-safe. Other optimizations are shown in the appendices.

5.3.1 Parallel C12 EOs

In a C1234 master, from the moment when a request is sent until the acknowledge is received, C2 can be computing something else. Analogously, C4 is overlapped with the interval starting when the request falls and ending when the acknowledge falls. Therefore, the C1234 has some parallelism within the LM.

On the other hand, with C12 and with C2 structures, we have parallelism between the first half of the handshake and C2, but during the second half of the handshake the LM has to remain idle, as shown in the next figure. Hence, we need to overlap the full handshake with C2. This overlap will substantially improve the performance only if the duration of C2 is similar to the time used by the communication: if the communication were much slower, it would completely dominate the throughput; if the communication were much faster, not much is gained by overlapping it with C2.

above the master/slave relation between LMs is fixed at design time. Nonetheless, for more flexible structures than those of systolic arrays it is desirable to have different LMs be masters at different times. The following analysis suggests that the mastership switch should be done one abstraction layer [44] above the EO hardware.

Suppose an EO consists of two LMs, which at some point will exchange mastership. The original master (LM1) initiates the transactions, and both machines operate in the top sub-loop shown in the following figure:
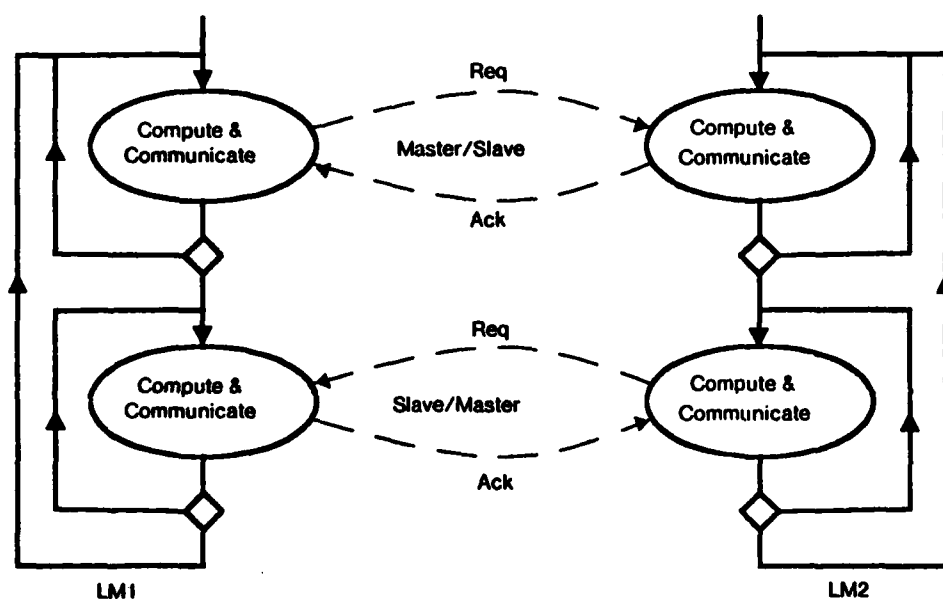
Figure 39. Mastership Switch between LMs

LM1 and LM2 can agree that after some event, both will make a transition to the bottom sub-loops. After the last acknowledge of LM2, LM2 becomes the master and will send requests instead of acknowledges. Therefore, LM1 will have to detect that LM2's last acknowledge has fallen, and some time t_{switch} later, that the first request from LM2 has arrived. Since the period t_{switch} is determined solely by LM2, LM1 must detect a pulse, instead of following the 2-cycle protocol.

Alternatively, at a given time, we can think of the LM that will wait next as the actual slave. In this way, the relative ordering of the edges of the req/ack signals is never changed, but the internal logic of the LMs re-interprets these signals. This is a safe way of transferring mastership, even though an observer looking at the handshake lines
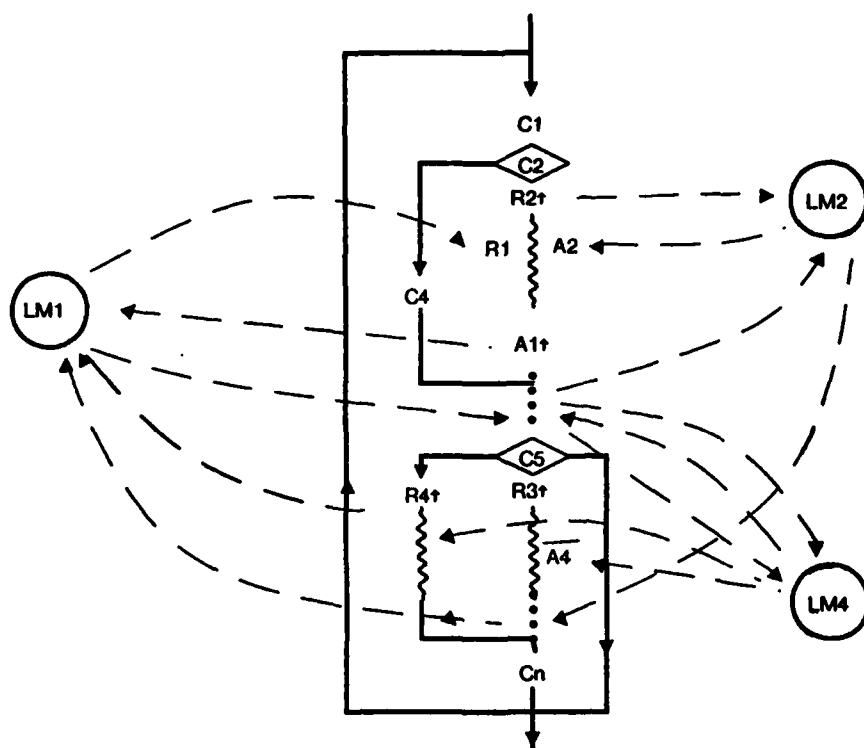
Figure 37. ESD for a Mesh of Interconnected LMs

Second, consider introducing some structure to the way in which LMs are connected. For example, we can assemble the grid shown in the figure below, which is suited for systolic arrays. Each LM is master of the LM to its right, and master of the LM below.
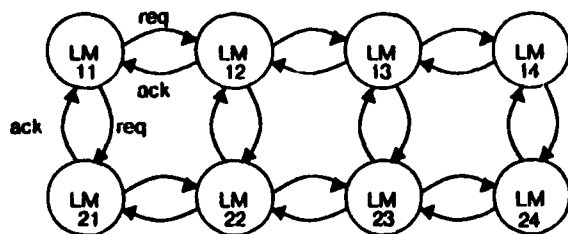
Figure 38. A rectangular grid EO

Finally, consider a more complex mastership relationship. Note that in the machine
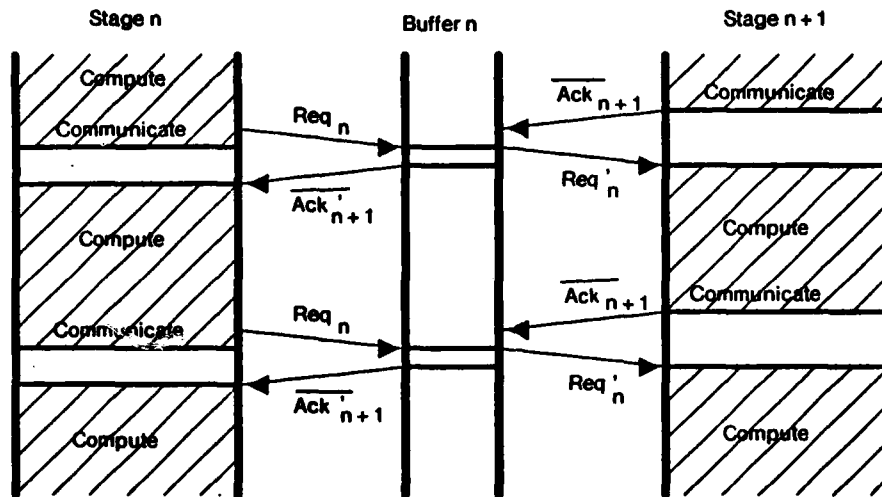
Figure 36. Improved Utilization Schedule of a C1 Pipeline with Interstage Buffers

5.2.4 Complex EOs

The EO structures we have discussed are fairly simple. They were assembled by connecting a sequence of basic EOs whose ESDs consisted of simple loops. The EO mechanism allows much more general structures. We can generalize the EO structures by using LMs whose ESDs have more than just the basic loop, by interconnecting the LMs in multidimensional meshes (instead of just one-dimensional pipelines), and by having the master/slave relations between LMs change while the EO is running. This section shows that the complexity of the resulting EOs can be substantial.

First consider an LM with a complex loop. When an LM has a basic ESD loop only, it is obvious that it will comply with a 2-cycle protocol; for more complex ESDs, all possible paths must also comply with the 2-cycle protocol to assemble an EO. As an example, the next figure shows a LM that is a slave of LM1 and master of both LM2 and LM4. The ellipses stand for parts of the ESD corresponding to computations and communications with other machines that were not drawn. Note that each LM can interleave the communications with many other LMs, since each C_i may contain not only computations, but also communications with other LMs.
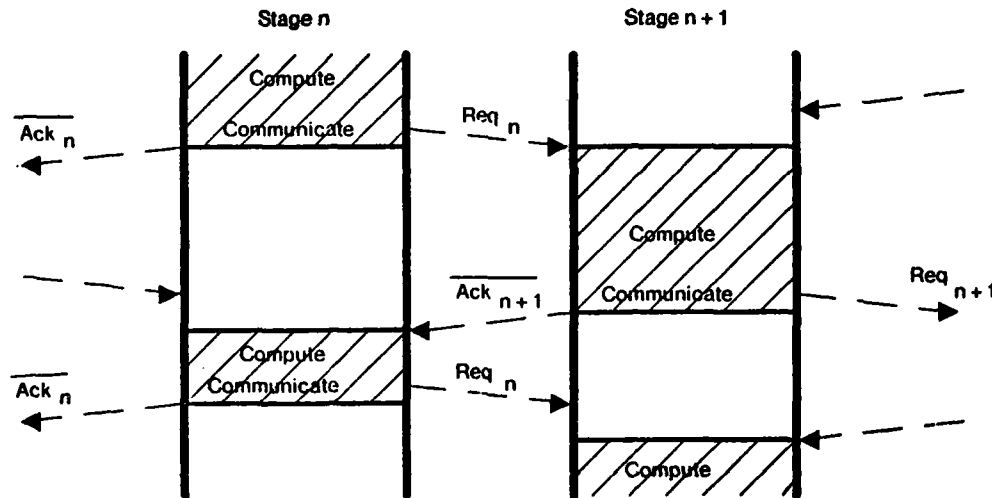
Figure 35. Operation Schedule of a Simple C1-only Escapement Pipeline

For this pipeline, at any time half the machines are computing while the other half are waiting. However, waiting consists just of stretching a clock phase. Hence, if the computational tasks are short and take a single clock cycle, actually all machines will be computing all the time, and no time will be lost. Remember that if the stretch line goes low before the normal end of a clock phase, the phase is not actually stretched, so the machines need not always delay each other.

If each C1 computation takes several clock cycles, the fact that odd and even stages alternately sleep wastes approximately half the potential computational power of the pipeline, as can be readily seen in the schedule above. If we interpose EO buffer stages in between the original stages, the stages and buffers will sleep alternately. The loss is now much smaller because the buffers are fast, while the stages take several clock cycles. Therefore, as can be seen in the next schedule, the efficiency of the pipeline can be improved considerably with buffering.
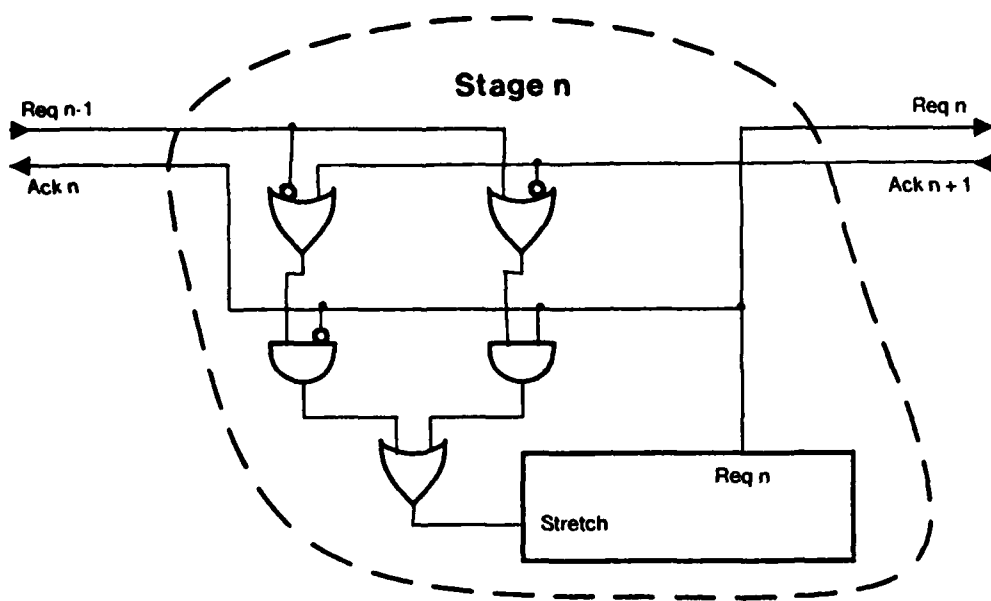
Figure 34. Stage of a C1 Simple Pipeline

To understand how the pipeline operates, assume that before entering the main loop of each stage, each LM has been initialized so that the *Req* and *Ack* flags have been cleared. Once in the main loop, each stage will wait for *Req* from its master and for $\overline{Ack}$ from its slave before proceeding with its corresponding C1. Once C1 is completed, the LM goes through the communication part of the loop, and eventually ends up back at the top of the loop. When the request from the left rises and the acknowledge from the right falls, it will start a new loop.

The behavior of the pipeline can be more easily understood in the following schedule, where the active periods are shaded and the idle periods are white. One loop through the ESD corresponds to one shaded segment in the schedule.

that satisfies the 2-cycle protocol may be connected safely to an EO, independently of its internal structure.) We can connect C1 basic EOs as shown below. In the following figures, note that the indices correspond to the stages, not to the interfaces. Also note that only the communication control is shown.
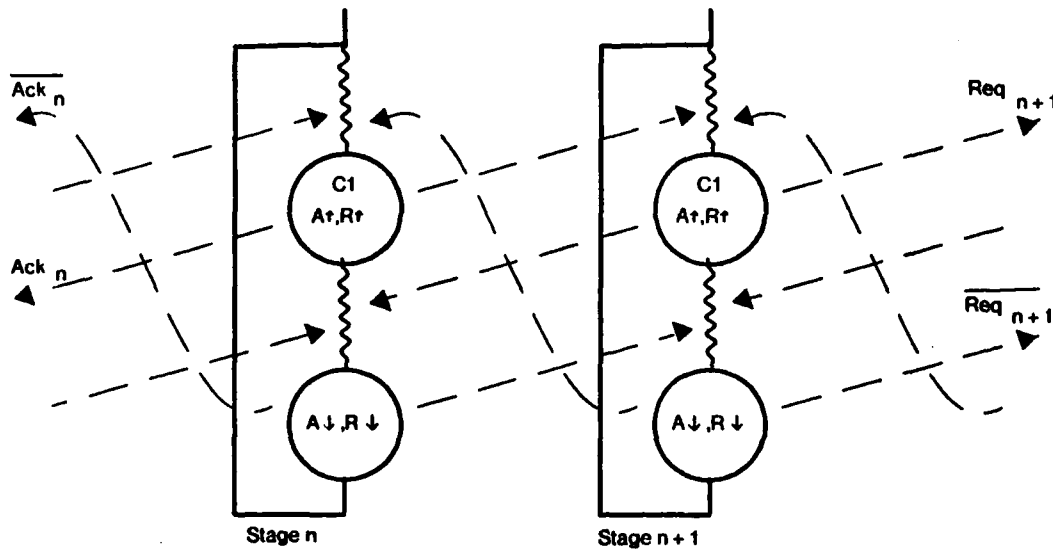
Figure 33. ESD for a Simple C1 Pipeline

For concreteness sake, we apply again the mapping rules to generate the logic diagram corresponding to a stage. The transformation is straightforward, and we can apply the mapping rules to the ESD mechanically, without having to think about the behavior of the machine at the circuit level: $Stretch_n = \overline{Req}_n(\overline{Req}_{n-1} + Ack_{n+1}) + Req_n(Req_{n-1} + \overline{Ack}_{n+1})$. Therefore, the logic diagram for each stage is:
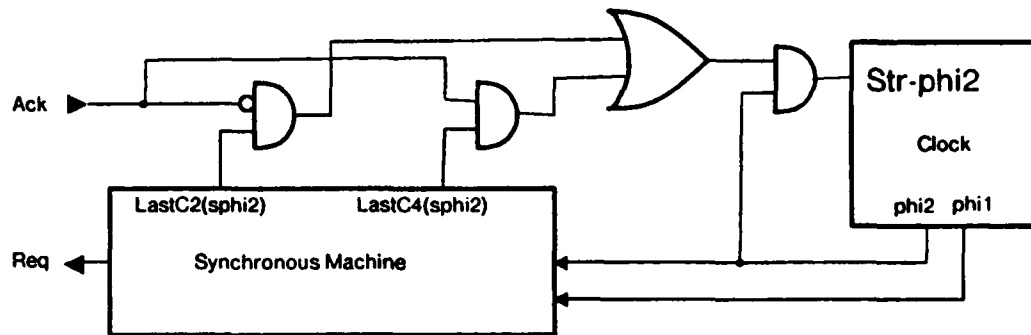
Figure 31. Master-Slave C1234 Circuit

Note that C1234 reduces to the SEM of the introduction if C2 and C4 are eliminated:
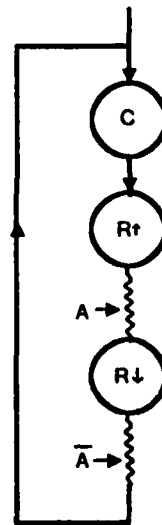
Figure 32. ESD of the SEM

Therefore, for SEM, the stretch condition of the Master C1234 reduces to just:

$$\text{stretch}\varphi_2 = (Req \cdot \overline{Ack}) + (\overline{Req} \cdot Ack) = Ack \otimes Req,$$

which is the stretch condition used in the logical diagram shown in the introduction.

5.2.3 Pipelines

To assemble a pipeline, simply connect a sequence of basic EUs.   (A machine

is naturally smaller than the set of arbitrary machines that can be described with logic diagrams, and value-safety of EOs specified with this formalism is decidable. This section first presents the specification notation, then the algorithms to verify value-safety, and finally the rules for mapping the specifications into efficient hardware.

5.5.1 A Language Extension for EOs

We already have a graphic notation in the ESDs, but it has several drawbacks. The ESDs get cumbersome for anything but the basic EOs. They are insufficient for the optimized EOs without interspersing logic diagrams, and they are inadequate for specifying EOs in a way that value-safety can be guaranteed. Therefore we will introduce a new, textual, EO specification language (EOL), whose purpose is to describe the EOs clearly and concisely, to guarantee value-safety by enforcing the protocols and the escapement mechanism for all asynchronous interactions, and to admit efficient compiling into efficient hardware.

To avoid re-inventing control languages [25] or more general hardware specification languages, we show how to extend any existing control language with our new EO-related constructs. The strict 2-phase clocking discipline (see Chapter 2) will be used for the LMs only to show clearly and concretely the interactions between the LMs within the EO.

Let the *"Cs"* of the ESDs stand for any expression or statement of the substrate language (SL) to be extended. Next, we describe each new construct of the EOL, show examples, and propose a syntax for them (which readers are welcome to change to suit their taste). For simplicity, assume that we use positive logic.

To describe the global relations between LMs, there is a single "binding" section for the whole EO, which specifies the connections of the handshake lines across LMs. Each handshaking line comprises a pair of request/acknowledge wires. The LM that appears first in each binding statement initiates the communication over the corresponding handshaking lines.

```
eo EOName;
bindings:
        mod1.handsh3 #mod2.handsh7,
        mod2.handsh2 #mod3.handsh1,
        mod1.handsh1 #mod3.handsh6;
```

Each LM emits (synchronously with the local clock) and receives (with combina-

tional logic that feeds into the stretch input of the local clock) handshaking signals. Hence, for each LM there is a declarative part in which the local name and type of each handshaking line is given. A "//" indicates concurrency of communication and computations (see the EO Optimization section above). For example:

> **modules:** ModuleNameList;
> **handshake** $v\varphi_1$: Data1, Data2, ..., GrantBus;
> //**handshake** $s\varphi_2$: OpenChan, CloseChan;

Each LM will be able to set, clear, and test some handshake lines; it is up to the SL to decide how to handle the rest of the synchronous operations. For setting, clearing or testing some line X, the following operators are used:

> X↑?   /* return TRUE if X is ON */
> X↓?   /* return TRUE if X is OFF */
> X↑!   /* set X */
> X↓!   /* clear X */
> X!   /* fork X */
> X?   /* join X */
> X⁇   /* join-fork on X */

For example, when a master and a slave communicate over line X, for the master X↑! is a $Req^\dagger$, while for the slave it is an $Ack^\dagger$. Whichever party waits for the other to set a line X will indicate it with X↑?. Note that this notation is consistent with the symmetric master-slave interpretation proposed when discussing mastership switching, where it was pointed out that the only difference between them is that the master opens the communication.

To make the mapping onto efficient hardware straightforward, we leave the designer the ability to indicate when each optimization discussed in the Optimization section should be used. For a parallel communication over line X, only two operators are necessary because the fork box handles the testing and clearing of X. Hence, it is necessary to indicate only when the parallel C2 may start the fork on X (X!) and when the computation has to wait for the communication to complete before proceeding after a join on X (X?). For packed join-fork, a single operator (X⁇) is enough.

5.5.2 Verifying Value-Safety of an EO Specification

To guarantee that an LM specified with the EOL can only result in a value-safe machine requires (a) global checking of the inter-LM bindings, and (b) local checking of the order in which handshaking lines are handled, to satisfy the communication protocols. Any problem related to the appropriate connection of stretch signals, choos-

ing the right phases, and other details of the escapement mechanism are also handled
by the compilation, so the EO designer has no way of introducing an error here.

(a) Each binding statement must have a different module name on each side (a master
and a slave). The compiler generates the corresponding request/acknowledge pair
of wires.

(b) For checking that communication protocols are satisfied, notice that no matter how
many interleaved communications with different LMs there may be, each one can
be checked by itself. To check a handshaking line X, it is sufficient that all possible
paths within the EO specification satisfy the communication protocol (see Chapter
2). For example, for a master on signal X, operators on X must satisfy the ordering
$X\uparrow!;\ldots$ $X\uparrow?;\ldots$ $X\downarrow!;\ldots$ $X\downarrow?;\ldots$, while for a slave on X they must satisfy the ordering
$X\uparrow?;\ldots$ $X\uparrow!;\ldots$ $X\downarrow?;\ldots$ $X\downarrow!;\ldots$. No path may contain only part of this sequence,
nor may control jump into the middle of a sequence, since either would clearly
violate the communication protocol.

5.5.3 Compilation of EO Specifications onto Hardware

Rules for mapping a verified EO specification into an efficient circuit are presented
next. "Efficient" means they result in circuits as compact and fast as those that can
be designed by a competent designer and similar to ones that have appeared in the
literature.

For simplicity assume that the signals generated by the synchronous control logic
(SCL) are v-φ_1, s-φ_2, and that the stretchφ_1 and stretchφ_2 inputs of the clock are
qualified inside the clock with φ_1 and φ_2 respectively. Rules 1 to 4 cover all machines
described in the section on structures for EOs.

R1.   Handshake declarations: For each handshake signal X there are two physical
wires, labeled X_{req} and X_{ack}, that are used to handshake with other LMs.

R2.   Outgoing handshake wires: For each handshake signal X, the SCL will have 3
associated outputs: X, $X\downarrow?$and $X\uparrow?$. X is connected to X_{req} if the LM is a master
on X, or to X_{ack} if the LM is a slave.

R3. $X\uparrow!$and $X\downarrow!$: For each set or clear command, the SCL will set or clear X.

R4. $X\uparrow?$and $X\downarrow?$: The SCL asserts $X\uparrow?$or $X\downarrow?$only for a single clock cycle. Let T be X_{ack}

for a master, and for X_{req} otherwise. For each $X\uparrow?$, stretch φ_2 when $[\overline{T} \wedge X\uparrow?]$, and for each $X\downarrow?$, stretch φ_2 when $[T \wedge X\downarrow?]$. Using R1-R4 gives machines such as the following:
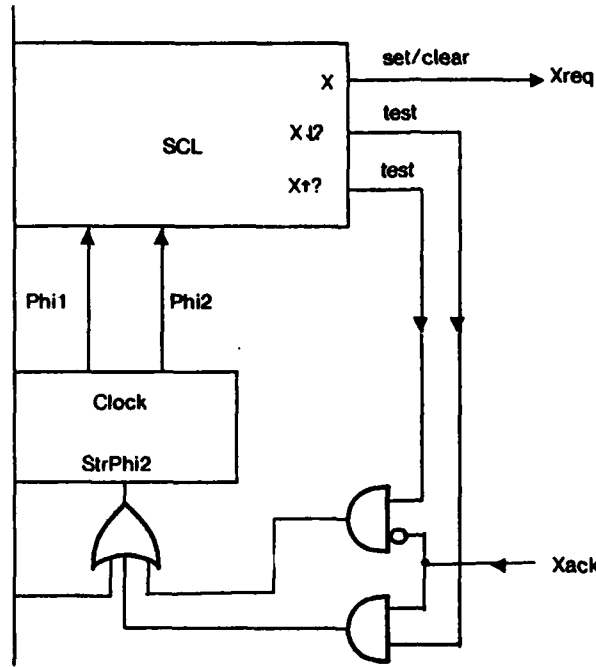
Figure 47. EO: master on X.

For the structures described in the Optimization section, Parallel C12 EOs subsection, rules R1 and R5 to R7 map the specifications onto circuits such as the one in the next figure. The SCL emits an X_{fork} signal that corresponds to SetReq for masters and to SetAck for slaves, and an X_{join} signal. For simplicity assume both are v-φ_1,s-φ_2.

R5. // handshake: For each parallel handshake signal X, there will be a fork box with the following pins: X_{set}, X_{clear} and X_{out}. For a master on X, connect $X_{out}\#X_{req}$ and $X_{clear}\#X_{ack}$. For a slave on X, connect $X_{out}\#X_{ack}$ and $\overline{X_{req}}\#X_{clear}$.

R6. X!: For each fork command, the SCL asserts X_{fork} for a single clock cycle. Connect the output of $[X_{fork} \text{ AND } \varphi_2]$ to X_{set}.

R7. X?: For each join command, stretch φ_2 when $[X_{join} \wedge End(X)]$, where $End(X) = [X_{ack} \vee X_{req}]$ for a master on X, and $End(X) = [\overline{X_{ack} \vee X_{req}}]$ otherwise. The SCL asserts X_{join} only for a single clock cycle.
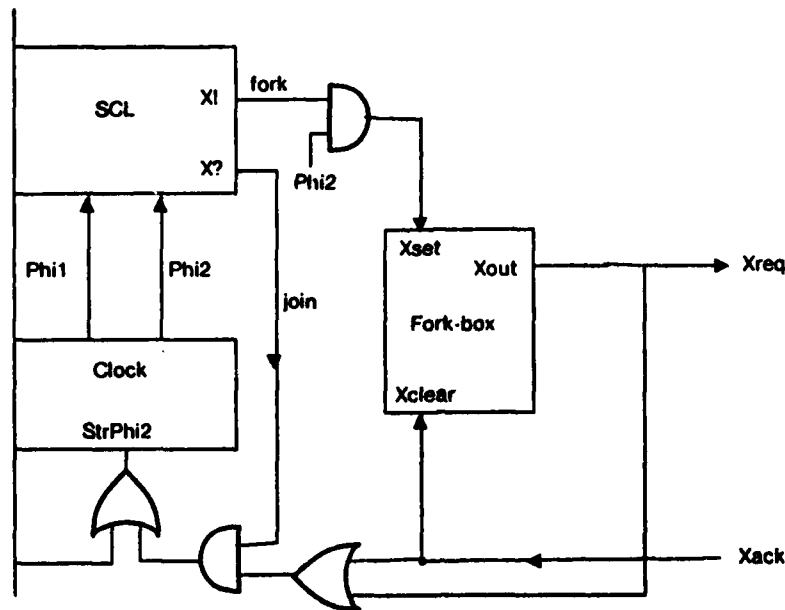
Figure 48. Parallel C12: master on X.

For the structures described in the Optimization section, Signal Packing in C2 EOs subsection, rules R1, R5, and R8 map the specifications onto circuits such as the one in the next figure. End(X) is used as in R7.

R8.   X?!: For each X?! command, the SCL asserts X?!-$v\varphi_1$,$s\varphi_2$ only for a single clock cycle. Stretch φ_1 when $[\text{X?!} \wedge End(X)]$ and connect the output of $[\text{X?!AND } \varphi_2]$ to X_{set}.
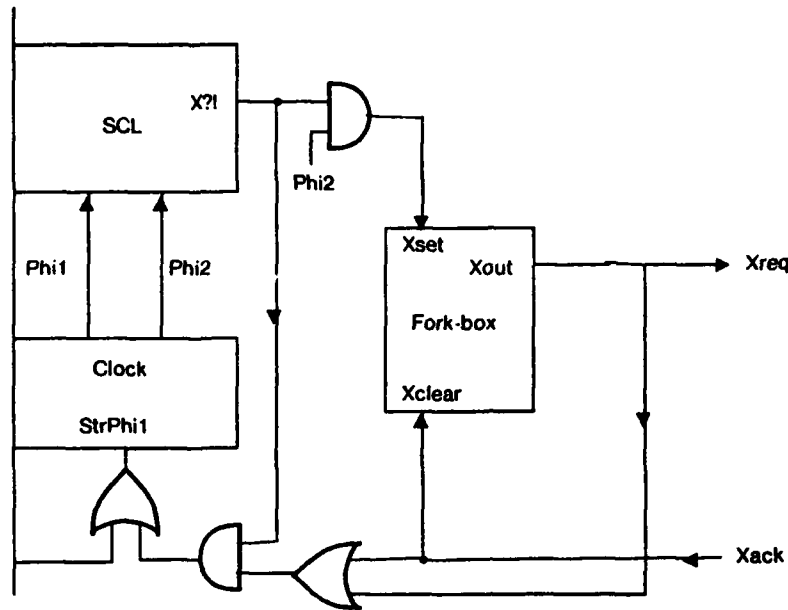
Figure 49. Signal-Packed C2: master on X.

5.5.4 Specifying, Verifying, and Compiling an EO: An Example

To recapitulate, we introduced the EOL to be able to guarantee the value-safety of EO circuits and to simplify their design. Also, we needed to avoid the notational complexity of the ESDs for the LM description for anything beyond the Basic loop, and in particular for optimization, where a mixed ESD/Logical notation was used, and also for the EO description, where even in the Simple Pipelines the multiple req/ack links needed for each master/slave relation were cumbersome. The EOL satisfies all of those requisites, and is used next to describe formally the behavior of Seitz's Pipeline Modules (SPM) [39]. Then, we verify and compile their specification.

Seitz's SPM is shown in the next figure. From the description of the SPMs and their circuit [39], the SPM behaves as an EO that is a slave of its neighbor to the left and a master of the one to the right. The clock runs all the time except when the EO needs input data that is not ready, or when it needs to output data but the output buffers are still in use from the previous transaction.
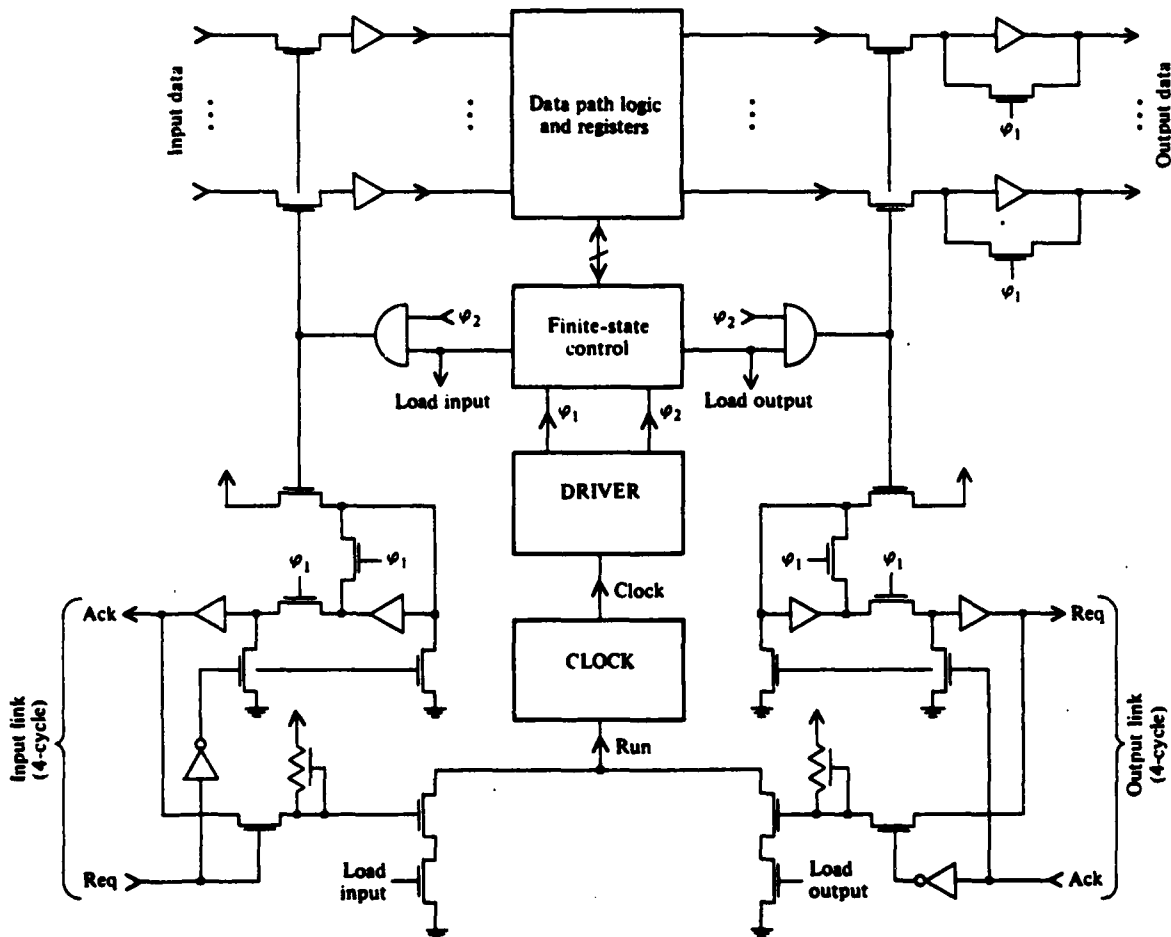
Figure 50. The original SPM (from [39])

Using the EOL, we can specify an EO that behaves like the SPM described above. Suppose an LM SPM2 has a master SPM1 and a slave SPM3, so the pipeline has 3 stages:

```
eo SPM;

bindings:
      SPM1.loadout # SPM2.loadin,
      SPM2.loadout # SPM3.loadin;

modules: SMP1, SPM2, SPM3;
// handshake, vphi1: loadin, loadout;
```

```
            Initialize;
            loop
                    {...;
                    loadin?;
                    ...;
                    loadout?;
                    ...}
            end
```

Figure 51. SPM Specification with the EOL

The verification is trivial, since there are only two independent parallel handshake signals, each of which is invoked once per loop with a join-fork command, satisfying a 4-cycle communication protocol.

Now, the middle module, SPM2, is compiled into a circuit using the mapping rules. For clarity we apply the rules only to the SPM2-SPM3 relation, since the SPM2-SPM1 relation is almost identical.

(1) From the binding section we find that SPM2 is a master of SPM3 on *loadout*, requiring 2 handshake wires (R1), labeled $loadout_{req}$ and $loadout_{ack}$.

(2) The SCL will output (R8) $loadout_{fork}$-vφ_1 , asserting it for one clock cycle when the SCL reaches the state where loadout? appears in the code.

(3) There will be a fork box (R5) connected as follows: $loadout_{clear}$#$loadout_{ack}$ and $loadout_{out}$# $loadout_{req}$.

(4) The output of [$loadout_{fork}$ AND φ_2] is fed to the $loadout_{set}$ input of the fork box (R8).

(5) Finally, the output of [$loadout_{fork}$ AND ($loadout_{req}$ OR $loadout_{ack}$)] is fed to the stretch-φ_1 input of the clock.

(6) Applying the same rules to the SPM2-SPM1 relation, we can obtain the slave side of the circuit, completing the SPM2 control shown in the following figure:
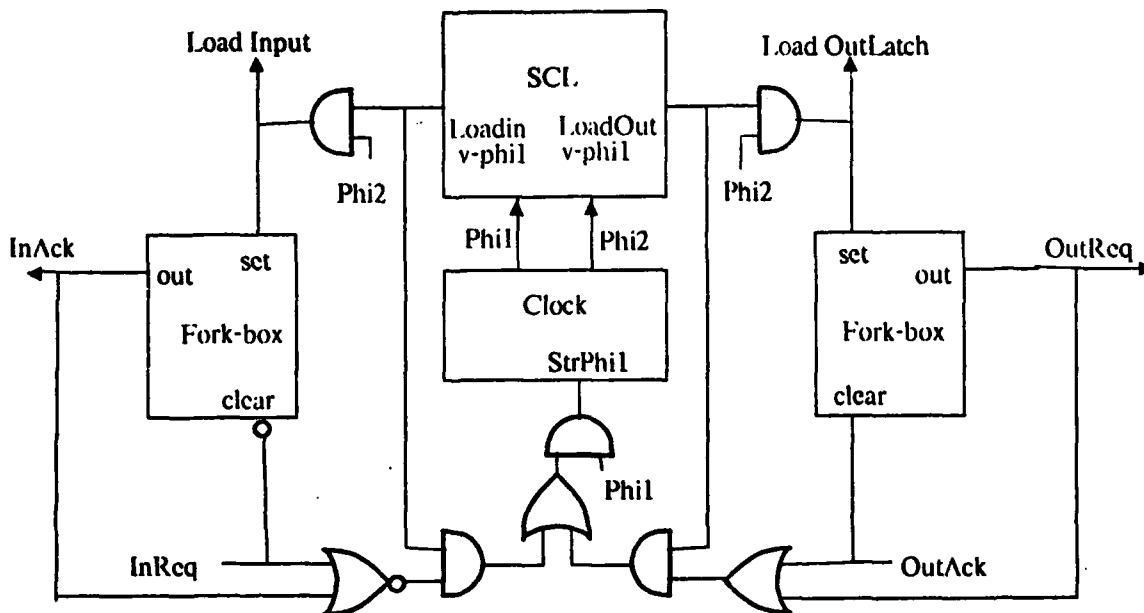
Figure 52. Compiled SPM

Notice that the compiled SPM compares favorably with the original SPM of [39]. It is a member of a general family of circuits, the EO specification can be modified and recompiled easily, and value-safety can be guaranteed. Also note that this circuit is very simple to construct using our synthesis technique, and that we could build more complex machines with ease.

§5.6 Summary

An architectural alternative for designing value-safe GA-LS systems without using synchronizers has been presented, and it was found that such machines have a variety and complexity that exceeds by far that of unsynchronous machines. The EOs were optimized to allow increased parallelism between computation and communication, and a specification language (EOL) was proposed. The EOL allows a designer to describe and modify the specifications of EOs easily, in spite of the possible complexity of the corresponding circuits. This specification can be automatically verified for value-safety, and then compiled onto compact and fast circuits.

The main limitation of EOs is that they cannot poll a multiple set of lines,

that their GA communication must follow a 4-cycle or a 2-cycle communication
tocol. When their performance was discussed it was seen that if the application
be accommodated by an EO structure, EOs can out-perform even unsynchronous
chines. EOs do not use synchronizers, cannot have unbounded stretching periods,
ond immediately to asynchronous requests, and restart their clock in phase with
ming requests.

Conclusion

1 Summary and Concluding Remarks

Synchronous systems cannot grow in complexity without limit because they have to interact with other components that cannot share the same clocking controls, because of delays in the communication across a system, and because of clock skews. However, if we partition a system all the way down to its simplest possible sub-components, the communication mechanisms between these elements would dominate the space and time used for the actual computations. As a consequence of these two opposing factors, we chose to use a GA-LS structure.

We analyzed the GA interactions in between LS machines, centering on the handling of completion signals and on two kinds of reliability, called value-safety and time-safety. This analysis led to a taxonomy of architectures, where we distinguished two general classes of value-safe GA-LS machines that we called unsynchronous (those that receive asynchronous signals on which there is no restriction whatsoever) and escapenent (those interacting asynchronously following particular communication protocols). The rest of the dissertation covered the analytic and synthetic problems posed by these two classes of machines.

The discussion of unsynchronous machines began with a study of a stretchable clock and a synchronizer with a metastability detector, which we then used to build value-safe GA-LS machines. We developed a probabilistic model to analyze quantitatively the trade-offs between reliability, throughput, and real-time constraints, and

ded that unsynchronous machines can run at higher speeds and with much reliabilities than equivalent synchronous machines using conventional clocks and onizers.

scapement machines take advantage of knowing in advance the direction in which ronous inputs can make a transition; they are value-safe without synchronizers. uiring that asynchronous inputs follow particular protocols, we developed EOs our advantages over unsynchronous machines: they require less hardware, since o not use synchronizers; they respond faster, since EOs do not need synchroniza-ock cycles; they respond in bounded time, since EOs have no metastability detec-wait for; and their clocks re-start in phase with the edges of incoming external ronous signals. EOs are structurally much more complex than unsynchronous nes, but we showed how to verify the value-safety of EOs and how to compile cifications onto efficient hardware.

the appendices, we prove several theorems relevant for the design of GA-LS nes. The DRV theorem proves it is not always possible to decide in a bounded it of time if an analog signal is above a given threshold. The Uncertainty theorem that a machine that solves a DRV problem cannot have certainty about both time ne values with which it operates. Finally, a theorem about the undecidability ue-safety for EO circuits shows that it is impossible to decide algorithmically er an EO circuit is value-safe. We also discuss in other appendices the design tchable clocks and show examples of our techniques in use in actual integrated s.

conclusion, this thesis provides a new theoretical and practical framework for sign of reliable, high-performance GA-LS machines.

ggestions for Further Study

he following paths may be worth pursuing:

rcuits and Processing: It would be useful to have faster and more compact designs stoppable clocks, possibly linking the speed controls of the clock to some process pendent parameter, so that a process resulting in faster gates would automati-ly result in a faster clock. It would also be interesting to study the phase-locking cuits needed to build unsynchronous machines with high-precision stretchable cks (Chapter 4) and the locking mechanisms needed to switch a stretchable

ock between internal and external frequency control (see the Performance and
eliability of EOs section in Chapter 5). Such work should address design mar-
ins, reproducibility, and yields for both stretchable clocks and synchronizers with
ietastability detectors (see Chapter 4).

lgorithms and Architectures: It would be interesting to experiment with nets of
Ms with various topologies (see the Complex EOs section in Chapter 5) and to
evelop algorithms that would take full advantage of the resulting architectures.
'he choice of appropriate granularity for both unsynchronous and escapement
iachines is an open question. It might be possible to perform a statistical analysis
o use the freedom given by the partial ordering of some tasks in the LMs, reordering
hem so as to maximize the throughput of the global system (see Optimization in
he appendices).

ystems and Networks: The discussions of the fundamental communication prob-
ims covered individual asynchronous signals and ways to handle them safely and
uickly (see Chapter 3). There are actually many classes of signals: e.g., controls,
ddressing lines, data lines, etc. It would be valuable to define systematically the
iteractions between these classes of signals given different delay and skew assump-
ions and also to develop protocols and circuits to provide convenient ways to link
tructured buses with the synchronous components of unsynchronous or escapement
ystems.

References

ns, R. et. al.: *"Real-Time Detection of Latch Resolution Using Threshold ns"*, United States Patent No. 3,515,993, June 1970.

wal, A. and Salz, A.: *"Mips-X: The External Interface"*, internal report, Nov.)omputer Systems Laboratory, Stanford University.

wala, T.: *"Putting Petri Nets to Work"*, IEEE TC, December 1979.

A. and Ullman, J.: *"Principles of Compiler Design"*, Addison Wesley (1978)

:au, F.: *"A Synchronous Approach for Clocking VLSI Systems"*, IEEE JSSC, SC-17, No.1, February 1982.

strong, D. et. al.: *"Design of Asynchronous Circuits Assuming Unbounded Delays"*, IEEE TC, Vol. C-18, N0.12, December 1969.

:acci, M.: *"An Introduction to ISPS"*, CMU-CS-78-137, Department of Computer ice, Carnegie Mellon University (1980).

, D. and Zehna, P.: *"Probability"*, Brooks/ Cole Publishing Co. (1971)

os, J. and Johnson, B.: *"Equivalence of the Arbiter, the Synchronizer, the h, and the Inertial Delay"*, IEEE TC, Vol.c-32, No.7, July 1983.

, I.: *"Time Loss Through Gating of Asynchronous Logic Signal Pulses"*, IEEE February 1966.

ley, J. & Rosenberg, F.: *"Characterization and Scaling of MOS Flip-Flop ormance in Synchronizer Applications"*, Caltech Conference on VLSI, January

END

FILMED

DTIC
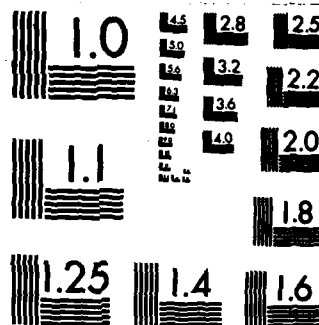
MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

[12] Chaney, T.: *"Measured Flip-Flop Responses to Marginal Triggering"*, IEEE TC, Vol. c-32, No.12, December 1983.

[13] Chapiro, D. and Mathews, R.: *"Clocking Cells"*, Chapter 3 of "The VLSI Designer's Library", Addison Wesley (1983), Newkirk, J. and Mathews, R. *(eds.)*

[14] Chapiro, D.: *"A Clocking Chip for Mips-X"*, forthcoming internal report, Nov. 1984, Computer Systems Laboratory, Stanford University.

[15] Chapiro, D.: *"Clock-Stretching Strategies for Mips-X"*, forthcoming internal report, Nov. 84, Computer Systems Laboratory, Stanford University.

[16] Chapiro, D.: *"Interrupt Handling in Mips-X"*, forthcoming, Computer Systems Laboratory, Stanford University.

[17] Couranz, G. and Wann, D.: *"Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region"*, IEEE TC, Vol.c-24, No.6, June 1975.

[18] Dennis, J.: *"A Preliminary Architecture for a Basic Data-Flow Processor"*, Computation Structures Group Memo 102, Project MAC, MIT, August 1974.

[19] Eichenberger, P.: forthcoming thesis, Department of Electrical Engineering, Stanford University.

[20] Feller, D.: *"An Introduction to Probability Theory and its Applications"*, John Wiley & Sons Inc (1950).

[21] Friedman, A.: *"Synthesis of Asynchronous Sequential Circuits with Multiple-Input Changes"*, IEEE TC, Vol. c—17, No.6, June 1968.

[22] Gurd, J. and Watson, I.: *"Data Driven System for High Speed Parallel Computing – Part 2: Hardware Design"*, University of Manchester, Computer Design, July 1980.

[23] Halpern, J. and Moses, Y.: *" Knowledge and Common Knowledge in a Distributed Environment"*, Proceedings of the 3rd annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Systems, 1984.

[24] Hanna, M.: personal communication.

[25] Hennessy, J.: *"Slim: A Simulation and Implementation Language for VLSI Microcode"*, Department of Electrical Engineering, Stanford University, California (1981).

[26] Hill, F. and Peterson, G.: *"Introduction to Switching Theory and Logical Design"*, John Wiley (1974).

[27] Hoare, C.: *"Monitors: An Operating System Structuring Concept"*, CACM, Vol 17, No.10, October 1974.

[28] Hollaar, L.: *"Direct Implementation of Asynchronous Control Units"*, IEEE TC, Vol. C-31, No.12, December 1982.

[29] Hurtado, M.: *"Dynamic Structure and Performance of Asymptotically Bistable Systems"*, D.Sc. Dissertation, Department of Electrical Engineering, Washington University, St. Louis (1975).

[30] Hurtado, M. and Elliot D.: *"Ambiguous behavior of logic bistable systems"*, Proc. 13-th Allerton Conf. on Circuit and Systems Theory, Oct. 75.

[31] Manna, Zohar: *"Mathematical Theory of Computation"*, Addisson Wesley (1974).

[32] Marino, L.: *"General Theory of Metastable Operation"*, IEEE TC, Vol. c-30, No.2, February 1981.

[33] Mathews, R., Watson, I. and Chenevert, D.: *"The Medium Tester Hardware Manual"*, Department of Electrical Engineering, Stanford University, 1984.

[34] Mead, C. and Conway, L.: *"Introduction to VLSI Systems"*, Addison Wesley, 1980.

[35] Nordman, B. and McCormick, B.: *"Modular Asynchronous Control Design"*, IEEE TC, Vol. C-26, No.3, March 1977.

[36] Noice, D.: *"A Two-Phase Clocking Discipline for Digital Integrated Circuits"*, PhD Thesis, Department of Electrical Engineering, Stanford University (Feb 1983).

[37] Pechoucek, M.: *"Anomalous Response Times of Input Synchronizers"*, IEEE TC, Vol. c-25, No.2, February 1976.

[38] Rosenberger, F. and Chaney, T.: *"Flip-Flop Resolving Time Test Circuit"*, IEEE JSSC, Vol. SC-17, No.4, August 1982.

[39] Seitz, C.: *"Asynchronous Systems"*, Chapter 7 of "Introduction to VLSI Systems" by Mead and Conway, Addison Wesley (1982).

[40] Seitz, C.: *"Ideas About Arbiters"*, Lambda, 1-st Quarter, 1980.

[41] Singh, N.: *"A design methodology for self-timed systems"*, Masters Thesis, Department of Electrical Engineering, MIT, MIT/LCS/TR-258 (1981).

[42] Stoll, P.: *"How to Avoid Synchronization Problems"*, VLSI Design (Nov/Dec 1982).

[43] Stucki, M. & Cox, J.: *"Synchronization Strategies"*, Caltech Conference on VLSI, January 1979.

[44] Tanenbaum, A.: *"Computer Networks"*, Prentice Hall Inc. (1981).

[45] Unger, S: *"Asynchronous Sequential Switching Circuits"*, John Wiley and Sons, Wiley-Interscience, 1969.

[46] Veendrick, H.: *"The Behavior of Flip-Flops Used as Synchronizers and Prediction of their Failure Rate"*, IEEE JSSC, Vol. SC-15, No.2, April 1980.

[47] Wann, D. et. al.: *"A Fundamental Problem Associated with the Physical Realization of Certain Classes of Petri Nets"*, Tech. Mem. No.215A, April 1977, Computer Systems Laboratory, Washington University, St. Louis, Missouri.

[48] Wormald, E.: *"A Note on Synchronizer or Interlock Maloperation"*, IEEE TC, March 1977.

Abbreviations

A/D: analog to digital.

Ack: acknowledge.

B: Boolean.

Cond: condition.

d: delay.

D: decay.

$\mathcal{DRV}$: decision on a real value.

EC: external clock.

EO: escapement organization.

EOL: specification language for escapement organizations.

ESD: extended state diagram.

f_d: data frequency.

f_c: clock frequency.

f_n: nominal clock frequency.

FSM: finite state machine.

GA: globally-asynchronous.

HC: harmonics-to-clear set.

I: input.

IC: integrated circuit.

L: loss.

LM: local machine.

LS: locally-synchronous.

ME: memory element.

MTBF: medium time between failures.

O: output.

q: qualified.

R: real.

Req: request.

s: stable.

SCL: synchronous control logic.

SEM: simple escapement machine.

SL: substrate language.

SN: storage node.

SPM: Seitz's pipeline modules.

T: throughput.

unsy: unsynchronous, unsynchronized.

unsyb: unsynchronous with bounded stretching.

v: valid.

vq: valid-qualified.

Appendix A: Stretchable Clocks

It is possible to construct variable-speed stretchable clocks as in [39] or by assembling a ring of modules as in Chapter 4. Nonetheless, such clocks are not all that convenient, and since stretchable clocks are crucial for unsynchronous and escapement machines, it is worthwhile to analyze them and to consider how to design and improve them in a systematic way.

Although digital clocks are normally designed using analog methods, we can abstract the key analog components by providing their functional description. Then different circuits can be built with these primitive blocks, and the global behavior can be determined without using analog methods outside the primitive components. We provide physical implementations for two such primitives; then we obtain the behavior of the clock from the solutions to equations that model the clock, using the functional description of the primitive elements.

§1 Primitive Elements: Delays and Decays

A delay element is the first primitive, and we use the notation $out = d(in)$. For concreteness, assume we build it using combinational logic, as shown in the next figure. Call τ the switching-time of a gate in the technology being used.
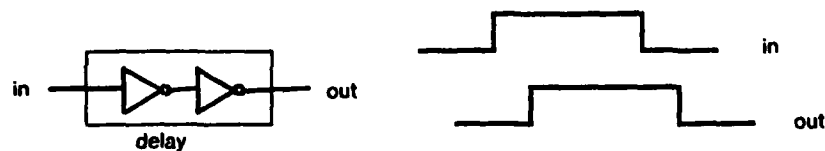
Figure 53. A delay element

When a digital input to a delay element changes value, the output will follow the input a time $|d|$ later, assuming the input transition time is of the order of magnitude of a few τ, and the input does not change again within a period smaller than $|d|$. If the input does not satisfy these assumptions, we do not care what happens with the output, since we will always satisfy them when modeling the clock.

The second primitive is a "decay", and we use the notation $out = D(in)$. For concreteness sake, assume we build it using the nMOS circuit shown in the next figure. A decay is an asymmetric delay that delays rising edges much less than it delays falling edges. Let $|D_{rise}|$ be the delay for $in\uparrow$, and $|D_{fall}|$ be the delay for $in\downarrow$. We assume that if an input pulse is shorter than $|D_{rise}|$, the output may be digitally undefined. Otherwise, input pulses will be stretched by an amount $|D_{fall}| - |D_{rise}|$.

We can implement a decay element by modifying an nMOS dynamic memory element: $|D_{rise}|$ can be fairly short (the propagation time through a few gates), and $|D_{fall}|$ is the time it takes to leak the charge from the storage node. To control the speed at which charge leaks, we use a pass transistor whose gate is controlled by an externally set analog voltage. This decay speed control allows us to change $|D_{fall}|$ over a wide range without affecting $|D_{rise}|$.
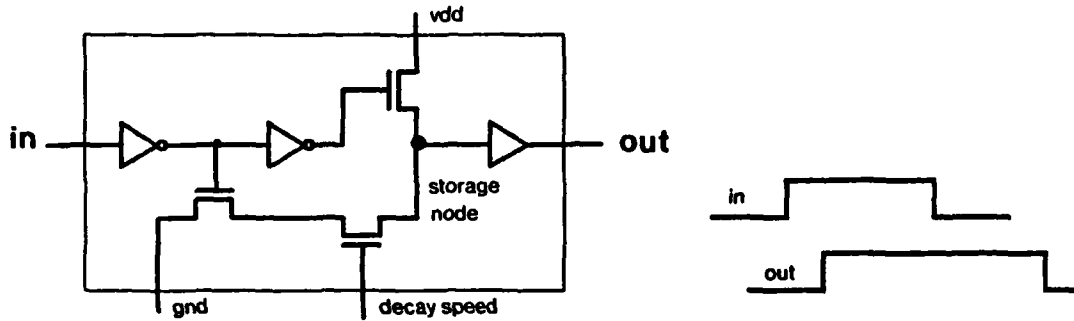
Figure 54. A decay element

§2 Clock Generation

We will build clocks by interconnecting almost identical modules. Each module, as well as the signals it emits, will be distinguished by a subindex. Given an ordered set of n modules, we define $\text{next}(i)$ as $mod_n(i + 1)$, *i.e.*, the next one in a ring of modules.

The approach we follow next is to use the delay and decay functions to state equations that have oscillatory solutions in the time domain. By analyzing these equations we will be able to derive most of the relevant properties and limitations of the corresponding circuits.

2.1 An inversion / delay ring

Even if it were possible to build inverters without delays, we could not use them to build a clock, as can be seen in equation (A.1), which does not have an oscillatory solution in time:

$$\Phi_{\text{next}(i)} = \overline{\Phi_i}. \tag{A.1}$$

Therefore, we must introduce delays: let d be the delay involved in computing an inversion. The equation for a normal ring oscillator is:

$$\Phi_{\text{next}(i)} = d(\overline{\Phi_i}), \tag{A.2}$$

which has an oscillatory solution, but only for n odd. To change the speed of the clock the delay through some modules must be altered (*e.g.*, Seitz [39] selects one of several alternate paths with different delays).

2.2 An inhibition / decay ring

There are a number of other equations that have oscillatory solutions, but we will go directly to the one we use for stretchable clocks. In the following equation:

$$\Phi_{\text{next}(i)} = \overline{\Phi_i} \wedge D(\Phi_i), \tag{A.3}$$

if Φ_i were high for a period longer than $|D_{rise}|$, $D(\Phi_i)$ would go high. Nonetheless, the inhibition path prevents stage $\Phi_{\text{next}(i)}$ from going high until Φ_i falls. Therefore, a stage goes high only after its predecessor goes low, and stays high for a decay period $|D_{fall}|$. A more complete equation (which takes into account the delay involved in computing the AND) would be:

$$\Phi_{\text{next}(i)} = d(\overline{\Phi_i} \wedge D(\Phi_i)), \tag{A.4}$$

which has an oscillatory solution for both and even n.

Since we have decoupled the rise and fall time delays, we can elongate a phase by increasing $|D_{fall}|$ without affecting the length of succeeding phases. If we take $n = 4$, we can obtain two non-overlapping clock phases separated by gaps and have independent controls for the normal length of each one of these.

2.3 An inhibition / decay ring with stretching

To introduce stretching, we must be able to extend the output of a stage for an arbitrary period (as long as some *stretch* signal remains asserted), as in the next equation:

$$\Phi_{\text{next}(i)} = [\overline{\Phi_i} \wedge D(\Phi_i)] \vee \text{stretch}, \tag{A.5}$$

which of course has the same oscillatory solution as equation A.4 when no stretch occurs. Clearly, if *stretch* is raised at arbitrary times, or has hazards, the solution to equation A.5 shows that these will show up in the output of the stage. Hence, we allow stretching of a phase *only* when its corresponding phase is already active (*i.e.*, stretch

ust be qualified $\Phi_{\text{next}(i)}$). Therefore the equation gets modified as follows:

$$\Phi_{\text{next}(i)} = [\overline{\Phi_i} \wedge D(\Phi_i)] \vee [\Phi_{\text{next}(i)} \wedge \text{stretch}_{\text{next}(i)}]. \qquad (A.6)$$

ι equation A.6, if *stretch* goes away before the normal end of the decay, it will not
ľect the phase length, but if it stays longer, the phase is extended. If the stretch
gnal had a dynamic hazard after the end of the decay, equation A.6 shows that it
ɔuld be propagated to $\Phi_{\text{next}(i)}$. To avoid this hazard, we require that stretch$_{\Phi_i}$ only be
llowed to make hazard-free transitions to zero during Φ_i, which is a requirement that
met by the stretching needs of both escapement and unsynchronous machines. Note
ι equation A.6 that $\Phi_{\text{next}(i)}$, as well as all succeeding gaps and phases, are not affected
ι any way aside from being delayed, no matter how long Φ_i is stretched (*i.e.*, no gap
r phase gets ever shortened or lengthened due to a stretching of a preceding phase).

.t this point, the partially developed clock looks as follows:
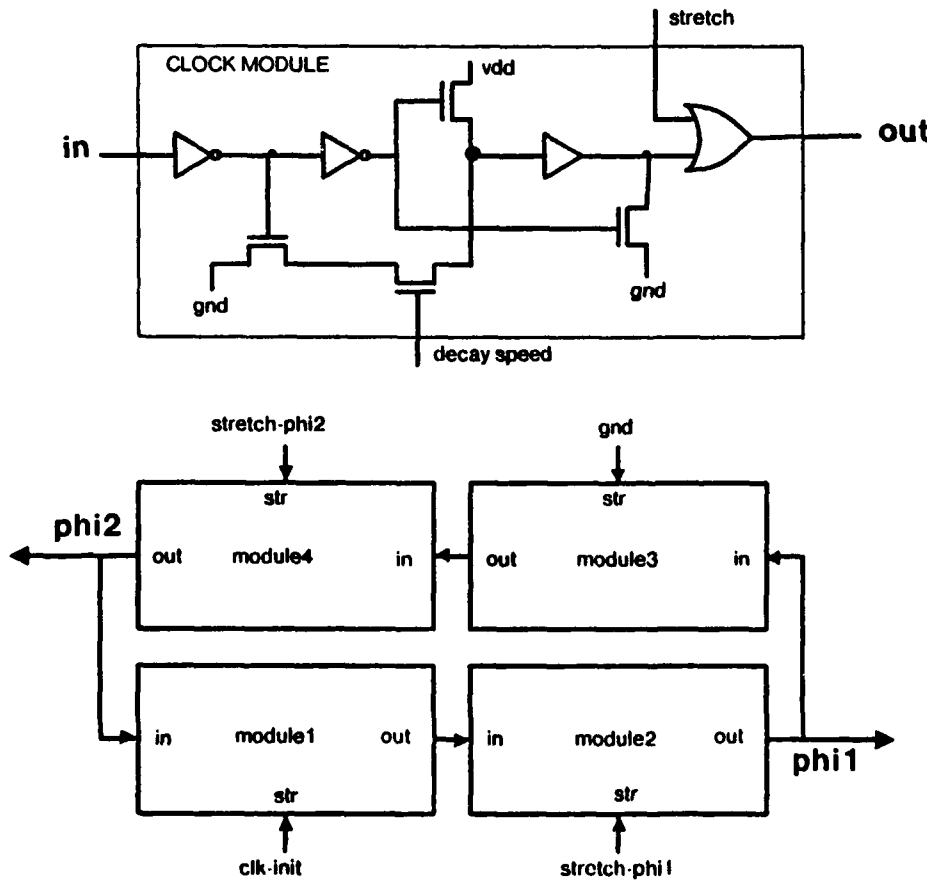
Figure 55. A Simplified Stretchable Clock

Pulsing one of the stretch lines for a long period starts this clock, and phases or gaps can be stretched for unbounded periods by asserting their corresponding *stretch* controls (*e.g.*, we stretch φ_2 by asserting the stretch input of module 4 with a signal that rises as a q-φ_2 signal). The normal length of phases and gaps can be set with their corresponding speed controls.

2.4 Inhibiting the Next Stage

In principle, it seems that the clock shown above is fine, and in fact it even works for a few days before harmonics appear, or the clock stops. To discover the reason and to cure these problems, we analyze equation A.6. What follows is a more complete version of equation A.6 that takes into account the relevant combinational delays:

$$\Phi_{\text{next}(i)} = [d_1(\overline{\Phi_i}) \wedge D(\Phi_i)] \vee d_2[\Phi_{\text{next}(i)} \wedge \text{stretch}_{\text{next}(i)}]. \tag{A.7}$$

Succeeding phases should not be high at the same time, but we can see in equation A.7 that $\Phi_{\text{next}(i)}$ will become high $|D_{rise}|$ time after Φ_i, *unless* the inhibition path (which takes $|d_1|$ time) is faster. Therefore we must satisfy:

$$d_1\uparrow < |D_{rise}| \tag{A.8}$$

for $\Phi_{\text{next}(i)}$ not to glitch.

2.5 Phase Length

Because of the inertial characteristics of the decay, it will filter out any pulse that is too narrow. Therefore, we must guarantee that:

$$|\Phi_i| > |D_{rise}|, \tag{A.9}$$

so that a stage detects that the previous stage has gone high and then low. In terms of energy, each stage must emit a pulse long enough to provide the energy to fully charge the next stage. Otherwise, the clock signal going around the loop of modules disappears, as can be experimentally verified by running the clock of [13] faster than this limiting speed.

itartup

n a ring oscillator, "all phases low" is an unstable condition (not a solution to rresponding equation A.2). Nevertheless, since $\forall i\{\Phi_i \equiv 0\}$ is a possible solution uation A.7, an explicit mechanism that will detect this condition and start the :hable clock is necessary.

f when the clock is not running one pulse is sent through one stretch line, equation ndicates that the oscillatory behavior will persist. Hence, we modify the equation iy changing it for only one value of i, that for convenience we take as 1.

$$i = 1: \quad \Phi_{\text{next}(i)} = \overline{[\overline{d_1(\Phi_i)} \wedge D(\Phi_i)]} \vee d_2(\Phi_{\text{next}(i)} \wedge \text{stretch}_{\text{next}(i)}) \vee \text{start}, \quad \text{(A.10)}$$

e *start* is obtained by NORing the values of the storage nodes of all the stages. lo not NOR the outputs of all the stages, because they never overlap. Note that : is no harm in having *start* be very long (it is a pre-start stretch), but if it were lse shorter than $|D_{rise}|$, the clock would not start properly. Hence, we make the ianism for detection of "all stages low" very slow.

Detecting and Eliminating Harmonics

For ring oscillators, harmonics may be unstable, but in the stretchable clock of. tion A.7, harmonics can persist indefinitely. That is, we may have

$$[\text{For } n > k > 1 : \Phi_i \wedge \Phi_{k+i} \neq 0]. \quad \text{(A.11)}$$

: that although the s ' 'n to equation A.7 with the initial conditions of equation i has no solution wher. '· harmonics are present, if somehow we manage to iduce harmonics, these are   able as the desired oscillation for $n > 2$. This lem can be verified experimentally with the clock of [13], when there is a power h. Therefore, we must check for all possible harmonics. Of course, there is no antee that after a power glitch the rest of the system will continue operating, but ast the clock must recover automatically.

We can talk of the harmonics in terms of how many modules apart (k) the waves traveling (*e.g.*, for the basic frequency, $k = n$). For $k = 1$, the inhibit mechanism ie modules prevents it. Every harmonic with $k > \lceil n/2 \rceil$, is detected as an $n - k$ nonic. Also, it is redundant to check for any harmonic k_1 such that it is a harmonic

.ust solve a $\mathcal{DRV}$ problem it cannot have both value and time safety.

'heorem 2. *No system that has components that have to solve $\mathcal{DRV}$ problems can ave both value and time certainty. That is, if such a system knows with certainty the alues with which it is operating, then it cannot know with certainty what the time is, nd vice versa.*

'roof :

(a) If the system always attempts to compute in bounded amounts of time (*e.g.*, it ses a free-running clock that provides its time reference), it is subject to the conse-uences of the $\mathcal{DRV}$ theorem, so we know it will have uncertainty about values.

(b) If the system doesn't require the $\mathcal{DRV}$ problems be solved in a bounded amount f time, it is not bound by the $\mathcal{DRV}$ theorem, but it will lose track of time. Since the time r the completion of the $\mathcal{DRV}$ problem is unbounded and cannot be pre-determined y any other part of the system, the component itself must generate a completion ignal C when it is done. It is not possible to poll C at regular intervals because of the orollary of the $\mathcal{DRV}$ theorem that says that sampling an asynchronous line is in itself $\mathcal{DRV}$ problem, nor can we receive interrupts because they are logically equivalent to olling a line. Since C is asynchronous with respect to all of the other signals in the ystem, any scheme that will *actively* attempt to decide whether the answer is ready will e-introduce the $\mathcal{DRV}$ problem. Hence, the only way to avoid checking synchronously r the arrival of C is to let the system become inactive at some point before C may rrive, and let it resume activity only when triggered by this completion signal. For he internal clock of the system, inactivity means that it must stretch one of its phases r gaps until C arrives. Call the resumption of activity after C "wake up".

If, on waking up, the system needs to know what time it is (or equivalently, ow long has the $\mathcal{DRV}$ computation taken), it cannot consult its own clock because he clock had to be stopped throughout the $\mathcal{DRV}$ computation, rendering its count of ime uncertain (we have absolutely no bounds on how much error it has accumulated). lence, the system must consult an external clock (EC) that didn't stop during this tretching period. EC will be asynchronous respect to C, so that when the system /akes up, its internal clock will also be asynchronous respect to EC. If the system ttempted to get the time from EC, it would find a synchronization problem that /ould re-introduce a $\mathcal{DRV}$ problem. Therefore, it cannot have certainty about the time nd remain value-safe. **Q.E.D.** ∎

se-locking, it is possible that when the phase-locking period has finished, the phases
· be completely out of sync.

"Solutions" to the DRV Problem

There are three main lines for impossible machines that attempt to solve the
chronization problem incorrectly. They deserve comment because there have been
nany erroneous solutions proposed with these common bases.

atches with different thresholds: two flip-flops with different thresholds r_1 and r_2
re used, with both thresholds contained within $[False_{max}, True_{min}]$. Hence, when
ampling a value, given the bounded slope at which the input may change, at most
ne of the flip-flops may go metastable. The problem with this "solution" is that to
hoose which flip-flop has the digital value is by itself another DRV problem, because
ι Boolean decision is being based on two inputs, one of which may be digitally-
ndefined.

Multiple sampling: with the same motivation, two samples of an input line are taken,
issuming that once a signal rises it will not fall until some acknowledging occurs.
Clearly, at most one flip-flop may go metastable, but again chosing the right one is
ι DRV problem by itself.

Using noise: since a flip-flop in a metastable state is not stable, a small noise signal is
ntroduced with the hope of forcing the flip-flop out of its metastable region quickly.
Noise, being random, may bring the system out of a metastable region, but it may
also drive it back into a metastable situation just when the system begins to evolve
owards a stable state.

Value and Time Uncertainty

If some components of a synchronous system must solve DRV problems, they will
ietimes generate digitally-undefined values that may cause problems throughout the
:em. Eventually, these values may either show up in the output, or may affect the
trol of the system itself if feed-back of any sort is involved.

Earlier we discussed value-safety and time-safety, and how to build value-safe
ynchronous machines that were not time-safe. Now we prove that if a machine

3.4 Corollary: Synchronization and Sampling of Digital Signals

Both synchronization and sampling of digital asynchronous signals involves decid-
ing if a digital asynchronous signal has arrived or not within a time window. It gets
reduced to the arbitration problem by looking at the local clock and the sampled signal
as two signals for which we have to decide which one arrived first. Within each clock
cycle we have to decide if we have seen a signal or not. If it has arrived within the
current clock cycle, it means it came before the clock's falling edge, but if it hasn't,
it means the clock's falling edge came first. Therefore, the arrival question is equiv-
alent to asking which of two signals arrived first, which is the arbitration $\mathcal{DRV}$ problem.
Hence, any source of asynchronous interrupts will sometimes get an acknowledge that
will be digitally undefined, and polling a line of an asynchronous device may result in
a digitally undefined reading.

3.5 Corollary: Phase Locking

To lock a recurring signal S to a reference signal R, it may be necessary to change
the frequency of S or its phase. Any action that retards the next zero-crossing of
S we will call "slow-down", while an action that advances it we will call "speed-up".
S and R are defined over a continuous domain (time) and can be initially displaced
with respect to each other by any time interval. Hence, to make the speed-up or slow-
down decisions, we must compute some real-valued difference function that provides
the necessary information to make the right Boolean decision, which is a $\mathcal{DRV}$ problem.
However, in phase locking, the ultimate result is not Boolean, so in principle we need
not make any Boolean decision yet.

Let a function $SP(\Delta\varphi)$ of the phase error $(\Delta\varphi)$, indicate when to speed up or slow
down. Stable points must have $SP = 0$. Since the machine does not distinguish phase
errors of multiples of 360°, $SP(\Delta\varphi)$ must be periodic with period 360°. Hence, at 0°
and at 360°, $SP(\Delta\varphi)$ must cross the zero axis with the same slope. Therefore, there
must be at least one other value $\mathcal{T}$ in $[0^o, 360^o]$ where $SP = 0$, which is a metastable
point.

Since it is not possible to solve the problem within the real domain, we are forced
to attempt a Boolean decision, by pushing the system in one or the other direction
when it is close to $\mathcal{T}$. However, to decide that the system is close enough to τ is
by itself a $\mathcal{DRV}$ problem. Therefore, no matter what clever device is used to achieve

3.3 Corollary: Arbitration

Arbitration involves deciding which of two (or more) asynchronous signals arrived earlier, so as to grant some resource to the sender of the first signal. Let R_n denote an n-dimensional real space, and B_n, an n-dimensional Boolean space. The two signals correspond to *two real* functions defined over the time domain, and granting the resource to one or the other requestor corresponds to a *single Boolean* value. Therefore, our problem has to map a value in R_2 into a value in B_1.

If we attempt to map R_2 into B_2, we find that since the arrival of each one of the two signals is asynchronous, just to decide that any one of them has arrived is equivalent to the A/D conversion problem: to decide that a continuous signal has arrived, we must define thresholds that allow us to make the distinction between "arrived" and "not-arrived". Since these thresholds define a DRV problem, determining that an asynchronous signal has arrived is also DRV.

Alternatively, we can first map from R_2 into R_1 with some function (RR) defined on R_1 whose values contain all the information necessary to decide which one arrived first (presumably some sort of difference function). No matter what transformation we chose, once we have this single, real-valued function, we still have to apply a transformation (RB) from R_1 into B_1 to perform arbitration. RB cannot yield constant answers, so there must be intervals in R_1 such that RB is True in some of them and False in others. These intervals must cover the target domain of RR, since RR must have the necessary information to produce the decision. The intervals cannot be overlapping because that would result in a conflicting answer. Hence, the intervals must be semi-open and contiguous, and must cover the target domain of RR. Each one of the real values (presumably a single one) that correspond to the boundaries between these segments marks a threshold that distinguishes the values in RB's definition domain for which RB is True from the ones for which RB is False. Clearly, each one of those thresholds defines a DRV problem for RB.

Hence, we can also apply the DRV theorem here, and say that any arbiter that must grant an available resource to the user who requested it first some fixed time after the request was made will sometimes grant it to nobody and sometimes to several users at once. What will happen depends on how will the digital users interpret the digitally-undefined signals that the arbiter will sometimes generate.

a $\mathcal{DRV}$ problem. Sometimes we will take shortcuts and simply show that some problem is equivalent to some other problem we already showed was $\mathcal{DRV}$.

Some of these corollaries have been already proved, but the proofs are much more complicated [32] or just limit themselves to equivalence proofs [9]. The reason our proofs are simpler is that we can use the $\mathcal{DRV}$ theorem, which does not assume that the circuit that solves the $\mathcal{DRV}$ problem is bistable, so we need not prove that a bistable is needed to solve a particular $\mathcal{DRV}$ problem.

3.1 Corollary: A/D Conversion

A one-bit A/D conversion corresponds exactly to the $\mathcal{DRV}$ class definition. Therefore, we can apply the theorem above and conclude that any one-bit A/D converter will sometimes require more time to produce a result than the time given to produce it.

If we have more than one bit, we have the same problem with any of the bits, no matter what encoding we use: for each bit that conveys information, there must exist some value T of the analog signal at which this bit will switch from being *off* to *on*. The switching value T defines a new $\mathcal{DRV}$ problem whose answer is given by that bit of the digital output. Hence, we should expect that any A/D converter will sometimes produce a non-digital output if it is given a bounded amount of time to sample an analog line and convert its value into a digital output.

3.2 Corollary: Schmitt trigger

Some of the "solutions" to metastability problems have attempted to use Schmitt triggers. Independently of the hysteresis cycle of a Schmitt trigger, if a signal that starts with its minimal value increases its value monotonically during an interval $[t_0, t_1]$, the Schmitt trigger will have to switch from *off* to *on* at its high triggering-point T_{high}. Throughout $[t_0, t_1]$ the output of the Schmitt trigger indicates whether the input is higher than T_{high} or not, which is a $\mathcal{DRV}$ problem. Analogously, T_{low} defines another $\mathcal{DRV}$ problem on decreasing ramps.

Since the proposed inputs fall completely within the domain of admissible inputs for a Schmitt trigger, we can apply the $\mathcal{DRV}$ theorem and conclude that there are input patterns for which a Schmitt trigger may switch half way and remain with an undefined output as long as necessary to cause trouble.

and $f(I > \mathcal{T} + \delta) > O_{True}$, where O_{True} and O_{False} are constants whose difference is bigger than the given ϵ.

Since the slope of f corresponds to the gain of the circuit, the slope must be bounded, and because of the continuity assumption, f must have a non-empty region within $[\mathcal{T} - \delta, \mathcal{T} + \delta]$ such that $O_{False} < O < O_{True}$. For this region, O cannot be mapped onto True or False because of the limited accuracy assumption. Therefore, no combinational circuit will produce only digital outputs, given an arbitrary input.

Graphically, this function also models a mechanical inverter as the one in the next figure, with a rigid lever. The input sets the position of the left arm of the lever, and the output is indicated by the right arm on the dial. In terms of the figure, there are positions of the arm for which obviously the output is not digital.

Figure 59. A combinational decision element

Therefore, we are forced to consider sequential circuits to solve the $\mathcal{DRV}$ problem. But, if we can solve the $\mathcal{DRV}$ problem in a bounded period using a sequential machine, then we could build with it a "perfect synchronizer" (one that cannot remain metastable for unbounded periods [9]) for a synchronous machine, contradicting Marino's theorem. Therefore, it is not possible to solve the $\mathcal{DRV}$ problem in a bounded amount of time. Q.E.D. ∎

§3 The $\mathcal{DRV}$ Class and the Corollaries of the $\mathcal{DRV}$ Theorem

The $\mathcal{DRV}$ class contains numerous problems of practical interest. We will show that problems belong to the $\mathcal{DRV}$ class by showing that their solution requires solving

Call *bounded* any change that is bounded *above* by some positive value, and also *below* by a non-zero value (*i.e.*, a two-sided bound). The discussion will refer only to voltages for the sake of conciseness, but in general we could reformulate the arguments using other physical parameters (*e.g.*, currents). The basic physical modeling assumptions that will be used are:

- Accuracy Assumption: For any *implementation* of a logic circuit there is a value δ such that the circuit *cannot distinguish* consistently and correctly between values less than δ apart.

- Continuity Assumption: Assume that voltages are everywhere *continuous and differentiable functions of time* (*i.e.*, voltage/time curves are smooth).

There are some fairly obvious implications of these assumptions. The first one is that different digital values must be some bounded voltage apart so that they can be distinguished as different values. The second one is that no observable value change can be instantaneous because different values must differ by a bounded amount and because of the continuity assumption. Third, the gain of any device must be bounded, because otherwise it would be possible to construct devices that violate the continuity assumption.

We will prove next that the basic assumptions lead to the impossibility of solving the $\mathcal{DRV}$ problem in a bounded amount of time.

- The $\mathcal{DRV}$ problem: given the assumptions of limited accuracy and continuity of values, measure a voltage and decide if it is above or below a certain threshold. An incorrect answer is acceptable for voltages less than δ apart from the given threshold, but the answer must be a digitally-defined value.

§2 Proof of the $\mathcal{DRV}$ Theorem

Theorem 1. *There is no machine that can solve the $\mathcal{DRV}$ problem in a bounded amount of time.*

Proof: Suppose that we attempt to build a combinational machine f that solves the $\mathcal{DRV}$ problem. Its circuit must map the input I and the threshold T onto an output O. To decide whether the output should be mapped onto True or onto False, given the limited accuracy assumption, the difference between O for $I < T - \delta$ and O for $I > T + \delta$ should be bigger than some given $\epsilon > 0$. Hence, we make $f(I < T - \delta) < O_{False}$

Appendix C: The DRV and Uncertainty Theorems

In this appendix we present proofs of the DRV theorem and the Uncertainty theorem. Theorems similar to the DRV theorem have been presented before [32, 9]. Here, somewhat weaker assumptions are made, so that the conclusions can be applied much more directly to a variety of problems. The Uncertainty theorem and the conclusions derived from it are new.

§1 Notation and Assumptions

As was discussed in Chapter 2, as far as the operation of bistables is concerned, Marino's paper [32] is conclusive in proving that sequential circuits exposed to inputs that can change asynchronously with respect to the activity of the circuit cannot avoid metastable conditions.

Our approach builds on Marino's theorem in the following way: we both assume non-anticipatory, continuous machines whose history can be subsumed (for the purpose of predicting future behavior) in the "state" of the machine. The main difference is that we do not assume our systems to be sequential machines, and that instead of proving that our systems cannot avoid metastability, we prove that they cannot make some kinds of decisions under particular conditions. When the DRV theorem is invoked in order to prove that some operation may produce metastability, it is not necessary to prove first that a sequential machine is necessary to perform the operation (as would be needed with Marino's theorem).

§2 Mips-X

Mips-X is a high-performance, pipelined, RISC (reduced instruction set computer), multi-microprocessor system, currently being developed at Stanford under the direction of John Hennessy and Mark Horowitz. Local caches and co-processors are used to in-crease the performance of the machine. To handle the interaction among the processor, the local caches, the co-processors and main memory, different variations of EOs (see Optimization of EOs in Chapter 5) are being considered [15, 2].

Since Mips-X is designed to run with a 2-phase, 20MHZ clock, and very short gaps, the stretchable clock can be made out of two stages. The automatic start-up mechanism (see Appendix A) has been made external to the clock, and since a 2-stages stretchable clock can't have harmonics (see Appendix A) the resulting CMOS version of the clock [14] is extremly simple (it has approximately one-third the number of gates of the clock shown in Appendix A). For frequency stability, the scheme described in Chapter 4 (High-Precision Stretchable Clocks) will be used [14].

stretch-bar   vdd   output

phi2

phi2

phi1

gnd   input   vdd   gnd
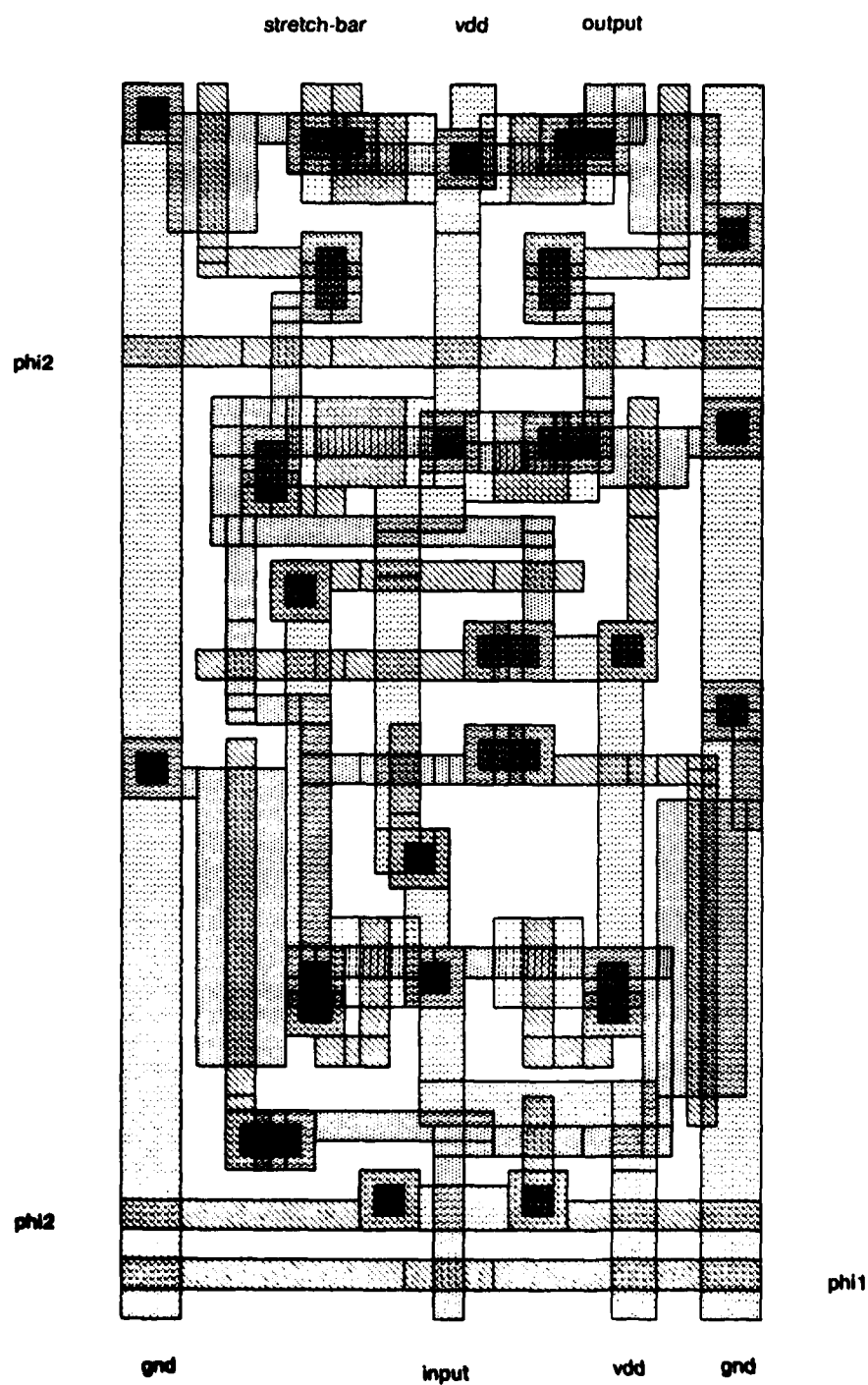
Figure 58. A Synchronizer with Metastability Detection

Appendix B: Experimental Machines

§1 The Medium Tester

The Medium Tester [33], is a functional tester for digital ICs, which is being used by several universities. The tester is implemented basically with three custom chips. One of these chips, the Test Controller, communicates asynchronously with an LSI-11 to exchange test vectors and other information necessary to set or test the values of the pins of the chip being tested. To attain a very high reliability, the Medium Tester has an unsynchronous architecture.

The Medium Tester uses the stretchable clock shown in Appendix A and synchronisers with metastability detection as the one shown in the following figure, which was also compiled and compacted using Lava [19]. This synchronizer samples an asynchronous input during φ_1 and may infrequently request a stretch of φ_2. The synchronized data is available in the output as a s-φ_1 signal. It pipelines the synchronizations, so that it can do one synchronization in each full clock cycle (for more details see [13, 39, 1]).
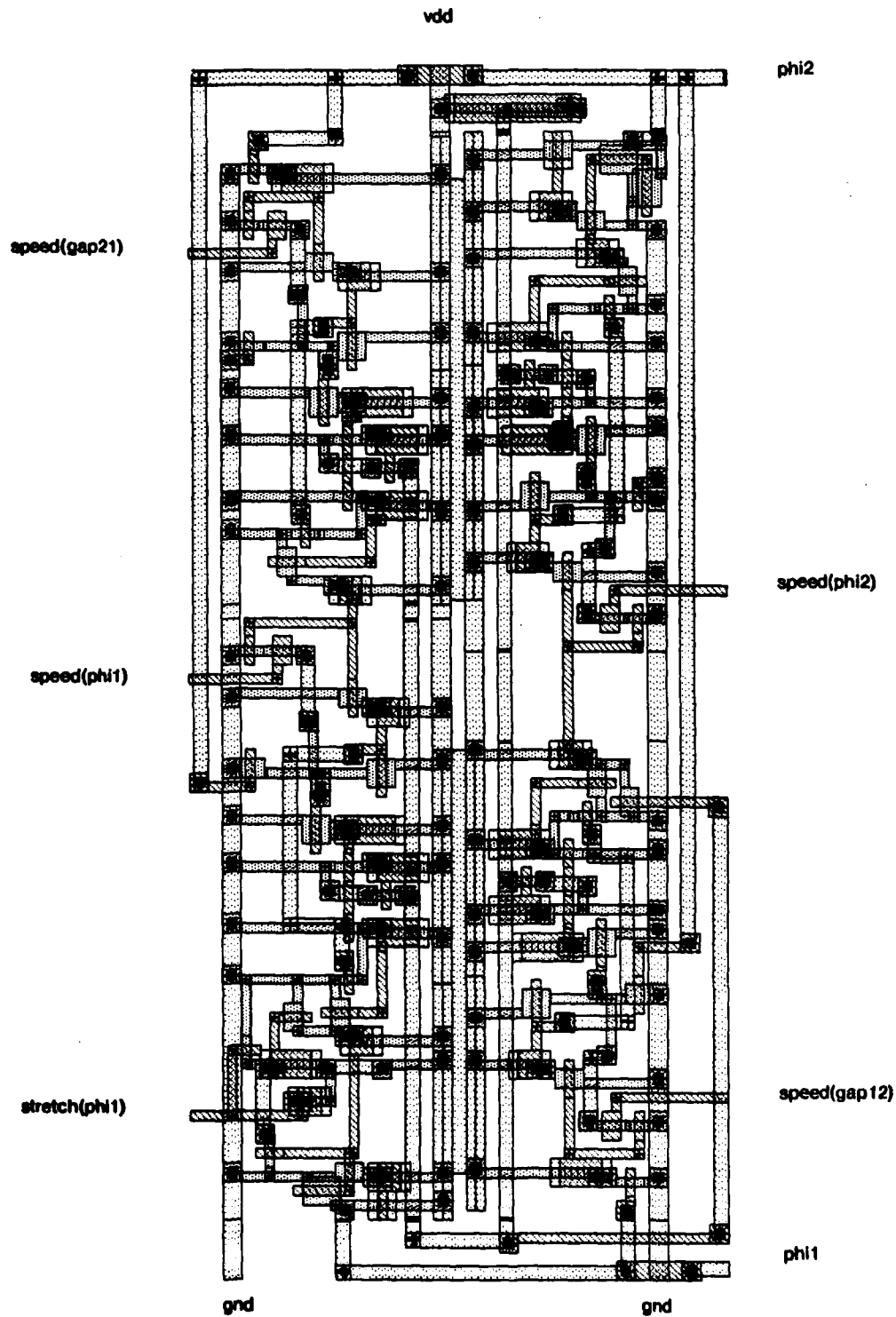
Figure 57. 2-Phase, Variable Speed, Stretchable Clock
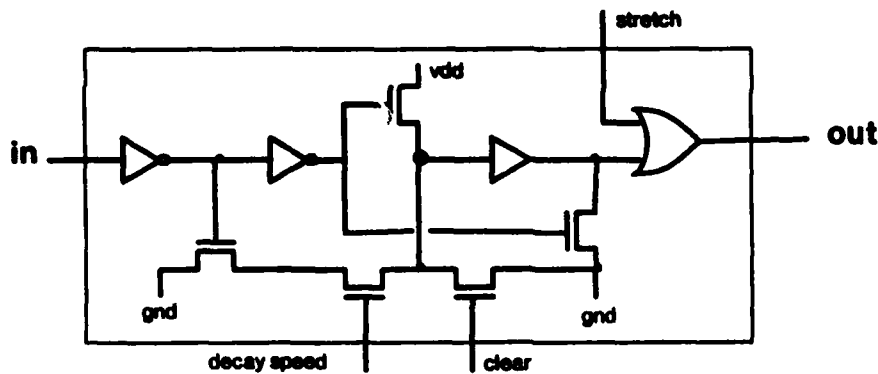
Figure 56. Clearing the Storage Node

§3 Implementation

Next we show an implementation of a 2-phase, variable speed, stretchable clock that incorporates harmonic detection and elimination and the automatic start-up features discussed above. This clock has been used in actual integrated circuits. The layout of this nMOS circuit was compiled and compacted automatically using Lava [19].

of another k_2 that is being checked. Therefore, we must only choose a pair of modules at a distance k, for the values that appear in the harmonics-check set, which is defined by the following recursive equation:

$$HC = \{i | \lceil \frac{n}{2} \rceil > i > 1\} - \{i | \exists \text{ a multiple of i in HC}\}. \qquad (A.12)$$

For a 4-modules unit, for example, we need to put a single check between modules 1 and 3, but for a 2-modules unit, no check is necessary.

Hence, for each value in HC, we chose a pair (i, j) such that $(j - i) \in HC$, and we define

$$\text{harmonic}_{i,j} = d_4(\Phi_i \wedge \Phi_j), \qquad (A.13)$$

where d_4 is the time it takes to compute the harmonic detection. It is important to note that if we extend the clock to n-phases, d_4 remains a constant because $|HC| < n/2$, so each of the n modules is used for at most one harmonic term.

In equation A.7 there is no way of making a harmonic solution vanish. Therefore, we must add a "clear" term to equation A.7, so that the non-stretching term becomes:

$$\Phi_{\text{next}(i)} = \overline{[\overline{d_1(\Phi_i)} \wedge D(\Phi_i) \wedge \overline{\text{clear}(\Phi_i)}]}. \qquad (A.14)$$

The remaining question is how to connect the "harmonic$_{i,j}$ detected" outputs to the "harmonic clear" inputs. Analyzing the solutions to equation A.14, we find that it is not enough to just clear either module i or module j, because this equation exhibits memory, so we need to clear either i and next(i), or j and next(j). This inconvenient memory effect appears because, although modules next(i) and next(j) are both low while harmonic$_{i,j}$ is on, their respective decay elements remember that their predecessors were high and attempt to go high as soon as the harmonic itself is eliminated. Therefore, we can chose to clear one or the other wave (i or j), but we must clear a stage and its successor, so we get:

$$\text{clear}(i) = \text{clear}(\text{next}(i)) = \text{harmonic}_{i,j}, \qquad (A.15)$$

where clear may be implemented by having in each stage an alternate, fast discharge path for the storage node of the decay, turned on by the "clear" signal, as shown in the next figure:

Thus, unsynchronous machines cannot be time-safe, and any scheme that attempts to bound their time uncertainty will destroy value-safety. It is also interesting to note that at any time, even though we *cannot know* the time with certainty, we *can know* what the time used to be when we first wanted to know it. This may sound puzzling, but it simply means that if we want to check the time shown by EC without risk of value confusion, we can use a synchronizer with metastability detection, connected to a local stretchable clock, as was shown in Chapter 4. But, when we get the time reading we do not know if now EC has produced a stretching of our internal clock precisely when it was trying to give us the actual time, so in fact we get to know with certainty what the time *was* when we asked for it, but not what *it is* at the moment when we receive it. It is important that we can know at least what the time used to be when we asked for it, because it means that our uncertainty about time corresponds to a single transaction, instead of accumulating forever.

Appendix D: Verification

This appendix discusses verifying the correctness of escapement and unsynchronous circuits. For verification purposes, it is convenient if one can describe the variety of signals that connect the different devices with some small number of signal types, state precisely in which ways these signals can be connected, and what are the resulting types of the combined signals. Types are associated to a few signals at design time, and then are propagated automatically to the rest of the circuit using the rules. Errors are found by detecting that invalid type combinations appear, during the propagation phase.

Verifying the "correctness" of a system, does not mean that "the system will work as expected". Correctness must be defined respect to some predicate. In our case, the strongest assertion that we will make is that if locally synchronous machines, verified correct respect to their specifications, are linked according to our rules, then the whole system will be value-safe. For convenience, we will assume the LMs follow a strict 2-phase clocking discipline (see Chapter 2), although we could use any other reasonable synchronous clocking method for the LMs. Since signals that indicate metastable conditions or control stretching of phases have no adequate counterpart in the strict two-phase theory, we will introduce new types.

There are some aspects that can be checked automatically very easily, and are not relevant here, so for clarity, we assume that there are no undriven nodes, that there are no dangling inputs or outputs, that there are no "fighting" conditions (different gates controlling the same node), that power has been appropriately connected, and that any necessary analog controls are connected correctly. Assume positive logic for simplicity. Furthermore, assume that at design time the components of each LM are tagged as

belonging to a particular LM, so that when we verify the global machine, inter-LM boundaries are obvious. For clarity, we will use φ_1 and φ_2 instead of talking about "one phase" and "the other phase". Obviously, the argument holds if we exchange all φ_1 for φ_2 and *vice versa*. When we say *connect*, we mean to join the lines making them a single electrical node; otherwise we use the word *combine* and we indicate what gates do the combining. When we talk about "gate logic", we mean combinational logic without pass transistors or any other kind of "switch logic". Node will stand for an "electrical node".

§1 Verification of Unsynchronous Machines

1.1 Types

- **Unsynchronized Signals:** Any signal X_{unsy} arriving from another clocking domain (generated by another LM).

- **Stretch Signals:** Stretch signals will be of type *stretch*, and will have a subtype φ_1 or φ_2 according to the phase that they should stretch.

1.2 Rules

A well-formed unsynchronous GA-LS machine satisfies the following rules:

- **Clocks:** Each synchronous sub-system has only one stretchable clock. The clock may have two input pins of type $stretch_{\varphi_1}$ and $stretch_{\varphi_2}$ These signals and the functional description of the clock provide the lowest level abstraction of the clock that the system designer deals with. The clock designer deals with lower level abstractions, which are discussed in another appendix.

- **Logic-Sense Propagation:** Given we know the logic sense with which stretch signals were generated by the synchronizer and the sense they are expected to have at the clock (both assumed positive logic), it is simple to propagate a sign with each stretch signal, such that this sign is inverted when the logic is inverting. It is then trivial to detect compatible inputs to gates and whether the final sense they have when they arrive at the clock is correct or not.

- **Connection of Stretch Signals:** They can be connected with other stretch signals with the same phase and sign.

- **Combination of Stretch Signals:** Stretch signals can be combined in a hazard-free way through gates whose Boolean simplification (taking the complete paths from their origin to the clock stretch input) must result in a single OR gate. Note that if the stretch logic diagram has a tree topology, it will be hazard-free. All stretch inputs to the same gate must be of the same phase and sign.
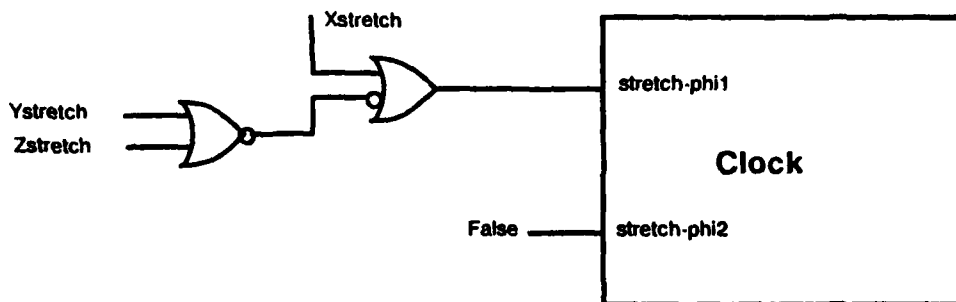

Figure 60. Stretch Combinations

- **Combination of Unsynchronized Signals:** They can be combined using any sort of logic with signals of strict types. The type of any such combination is unsynchronized.

- **Synchronization:** Each sub-system may have any number of synchronizers, each of which can take one unsynchronized input. The sampling control of the synchronizer will be a *qualified* signal. The synchronized output will be *stable* on the qualification phase of the sampling signal, while the generated stretch signal will have a complementary phase.
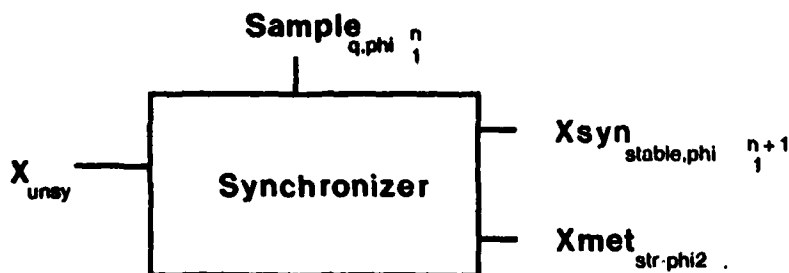

Figure 61. Synchronizer Types

1.3 Value-Safety of Well-Formed Unsynchronous Systems

Whenever a signal has to cross a boundary across synchronization domains, when it enters the other machine, it is fed to a synchronizer that transforms its *unsy* type to a *stable* type. There is no need to connect *stretch* signals of other types. Hence, we only check that they are combined correctly with other *stretch* signals, so that the machine will proceed only when all synchronizers are stable.

Therefore, each LM works synchronously, and the synchronizer/clock encapsulation, given by the functional definition of the clock and the synchronizer and by the combination rules, guarantees that any external signal that an LM may need will acquire the appropriate strict type, thereby preserving the value-safety of the system as a whole.

§2 Incompleteness of Verification for EOs

In this section we show that there is no general way to guarantee safety for EOs by just analyzing the circuit. Although we saw that we can verify the value-safety of unsynchronous circuits, we will see this is not possible for EO circuits.

EOs verification is harder because the clock stretch input is no longer shielded by a fairly limited number of possible combinations of outputs from local synchronizers. Instead, the clock stretch input is controlled directly by logic that combines external and local signals. Just to introduce a note of optimism, remember that the synthesis mechanism for EOs (Chapter 5) allows us to generate EOs that are value-safe by construction, so this section is mainly of theoretical interest.

Theorem 1. *Rules Systems for Safe EOs: There is no algorithm that can guarantee value-safety or time-safety for escapement organizations, given a description of the circuit, i.e., we cannot write a program that will tell us whether arbitrary EO circuits are value-safe or not.*

Proof: The *stretch* inputs of a clock are formed by a conjunction of an external stretch condition and an internal signal that indicates that the system has reached a state at which this stretch condition is allowed to cause stretching. If the internal signal happens to rise at some incorrect time, there are two possibilities that produce problems:

(a) The external condition will drop after some action by the LM, but the LM is sleeping, and the system deadlocks. Hence, even if it is value-safe, it doesn't work.

(b) From the semantics of *Req* and *Ack* signals we normally know the transition direction of stretch signals, but now things may be out of order, and we no longer have certainty about the direction of the transition of the external condition for the stretch. If *Cond* happens to rise by the *end* of the phase that was to be potentially stretched, we have a race between the end of the clock and *Cond*. If such a race occurs, the stretch inputs to the clock may no longer be dynamic-hazard free, thereby fouling up the clock operation. The clock now may glitch or stay in an incorrect stretch. Hence, we cannot count on any subsequent digital operation in the system nor on the clock to continue operating, so we lose both value-safety and time-safety.

The key issue is that we cannot rule out these situations by just looking at the circuit because finding that some flag will *not* be set at the wrong time is equivalent to Turing's Halting Problem.[1] Hence there is no general method to verify arbitrary EO circuits and guarantee that they cannot enter a deadlock as in (a) or a metastable state as in (b). Q.E.D. ∎

2.1 Verification of Well-Formed EO Circuits

It was seen in the theorem above that the rules for circuit descriptions are per force going to be incomplete for verifying them. Therefore, we might attempt to circumvent this theorem by solving a simpler problem: we request the designer to supply assertions about the EO. The designer would have to guarantee the correct operation of the inter-LM communication protocols that depend on the programs running on the synchronous LMs and a consistent interpretation of each inter-LM signal as regards to meaning and logic sense. Hence, we would be simply passing the critical problem to the designer and not solving the key issues. What would be left for the verification would be to prevent things like mistakenly connecting an address line to a handshake line, making sure lines are not left dangling, qualifying stretch lines on the appropriate phases, etc. Although this is far from guaranteeing safety, it might reduce the possibilities for errors.

---

[1]Since the EO is a FSM with unbounded I/O, it can simulate a FSM with two unbounded push-down stacks, which is equivalent to a Turing Machine [31].

In this sense, such rules could be thought of as the syntactic checks of a strongly typed language, which catch many trivial errors at an early stage. Nonetheless, the synthesis mechanisms we discussed in Chapter 5 dominate this alternative in every respect, because they can guarantee value safety and because they simplify the design process considerably without a loss in performance.

§3 Conclusion

We have seen that it is possible to develop rules for the verification of value safety for unsynchronous circuits, but that this cannot be done for general EO circuits. Nonetheless, since we have developed synthesis algorithms for EO circuits that generate efficiently value-safe efficient EO circuits from higher level specifications, we do not attempt to go around this theoretical result, and restrict verification of circuits to unsynchronous machines.

Appendix E: More Escapement Optimizations

The following optimizations speed up the EOs, or simplify their circuitry, by modifying their ESDs. As long as the partial ordering imposed by the communication protocols is satisfied and the application-dependent constraints are satisfied, both handshaking operations and computations can be moved to other places in the ESD. In particular, if each LM has several independent relations with other LMs, there may be many tasks that can be freely reordered.

§1 Flag Merging

Flag merging is a modification of the ESD that results in an area-speed tradeoff. Although the arrows pointing to different squiggles of a given LM can be handled independently, if we merge some of the arrows, we can reduce the number of signals emitted by the synchronous control logic (SCL) because for each squiggle the SCL must emit a *last* signal indicating the completion of the task preceding the squiggle.

Call each state transition (arc or squiggle) a *step*. Call the portion of ESD between two steps (S_1 and S_2) a *program*. We will say that S_1 and S_2 commute ($S_1 \circ S_2$) if the semantics of these steps allows to commute their order without altering the functional specification of the EO. No operations on the same handshaking signal can be commuted; for all other situations, the designer must know if a commutation is possible. We will say that a step S and a program P are commutative ($S \circ P$) if all steps in P are commutative with respect to S.

Clearly, an EO remains functionally invariant if a step S is displaced to another place in the state diagram if the program P defined by the old and new positions of S is commutative with respect to S.

Let X and Y be two independent handshaking lines, and call the two programs defined by $[X \uparrow, Y \uparrow]$ and $[X \downarrow, Y \downarrow]$, $P \uparrow$ and $P \downarrow$, respectively, as shown in the figure below. If $(X \uparrow \circ P \uparrow) \wedge (X \downarrow \circ P \downarrow)$, then $X \uparrow$ and $X \downarrow$ could be displaced to where $Y \uparrow$ and $Y \downarrow$ are, respectively. Hence X and Y can be merged into a single flag:
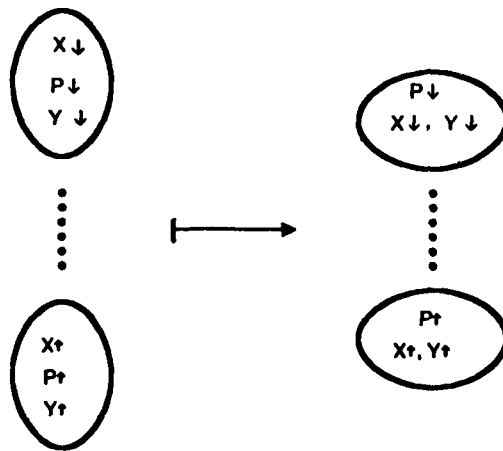
Figure 62. Flag Merging

The transformed LM will have fewer flags to handle, so that it will be smaller and simpler, but the global EO may be slowed down. For example, the modified LM may delay a response because it has to complete some unrelated task before it can send the common (merged) completion signal. For example in the simple pipeline of Chapter 5, each LM sent a request to the right and an acknowledge to the left, which were merged into a single signal, thereby making the machine slower but smaller.

§2 Stretch Merging

We saw that flag merges simplify the LMs, but may slow them down. Stretch merges do not change the logical complexity of the LMs, but may speed the EO by delaying to the last possible moment the checking of conditions that may result in stretching a phase. The LM can overlap useful computation with the time required for the external stretch condition to disappear.

Let S_1 and S_2 be two stretch conditions associated with two different squiggles in the ESD, and let P be the program defined between t he two stretch conditions. If $(S_1 \circ P)$, then S_1 can be computed (checked) in paralle l with S_2, as can be seen in the next two transformations. First, we can commute S_1 and P, and the two stretch conditions will end up one after the other:
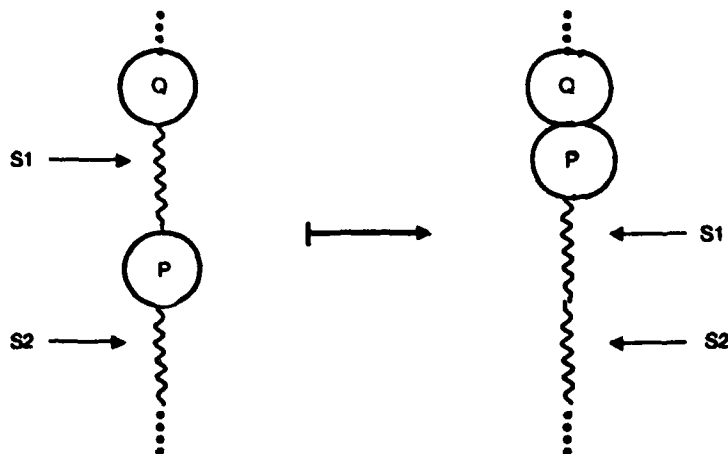

Figure 63. Stretch Moving

Since the two stretch conditions succeed each othe r without any intermediate computation, they share the same "last task" before th e stretch, and consequently, the conjunction with the internal condition for stretchin g will be the same for both squiggles:
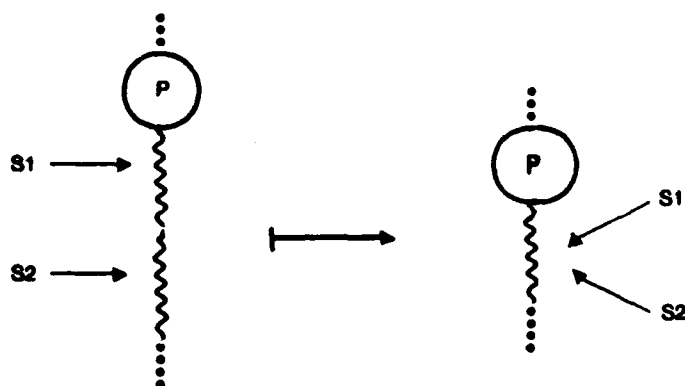

Figure 64. Stretch Concurrenc y

Hence, the previous transformations yields the stretch condition:

$$(last_Q \wedge S_1) \vee (last_P \wedge S_2) \;\mapsto\; last_P \wedge (S_1 \vee S_2).$$

The LM waits for S_1 only after the completion of P, instead of waiting before computing P.

§3 Replacing Squiggles by Arcs

It is possible to suppress altogether some stretch squiggles when the stretch will never actually occur.   For example, suppose that LM1 can guarantee that it will always respond to LM2 so fast that whenever LM2 goes to check for the reply, the reply has already arrived. Then LM2 need not stretch-wait for the reply, and instead it can proceed directly to the following state. In terms of the ESD, we have replaced a squiggle-arc by an arc, and in terms of the logic diagram, we have eliminated a few gates.

END

FILMED

7-85

DTIC